Augmenting Type Inference with Lightweight Heuristics

Inauguraldissertation
der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von

Nevena Lazarević

von Serbien

Leiter der Arbeit:
Prof. Dr. O. Nierstrasz
Institut für Informatik
Universität Bern

Von der Philosophisch-naturwissenschaftlichen Fakultät angenommen.

Bern, 21.06.2017.

Der Dekan:
Prof. Dr. Gilberto Colangelo

This dissertation can be downloaded from scg.unibe.ch.

Attribution–ShareAlike

To my parents who always support my every decision.
To my husband who always understands me.

Mojim roditeljima koji me uvek u svemu podržavaju.
Mom suprugu koji uvek ima razumevanja za mene.

Acknowledgements

First of all, I would like to thank Oscar Nierstrasz, for giving me the opportunity to work as a part of the Software Composition Group. I am eternally thankful to him for guiding me through the PhD life, for encouraging the research I was doing, and for giving desperately needed advice whenever necessary. I really enjoyed being a member of SCG, and solving every one of the coffee puzzles!

I am very grateful to Stéphane Ducasse, first and foremost for agreeing to be on my PhD committee, and for providing me with his comments on my work. I thank Gerhard Jäger for chairing the PhD committee.

A big thank you to all of the guys from SCG, who shared their time with me. Thank you, Mircea, for making every day shiny, joyous and colourful, even when it poured like hell. :) Thank you, Claudio, for finding time to read my work numerous times, and especially for sharing the office with me for, hm, eight months. :) Thank you, Haidar, for being a good friend, giving me tips for my travels and singing with me. Thank you, Boris, for all of our discussions in Serbian. Thank you, Andrei, for always asking how my research is going. :) Thank you, Andrea, for always smiling and being positive. Thank you, Jan, for always having a calm word and a piece of advice. Thank you, Leonel, for your precious advice regarding my trips to Chile and Argentina. Thank you, Mohammad, for always forcing me to perform better, even when I did not believe it was possible. :) Thank you, Yuriy, for all the laugh during coffee breaks. Thank you, Manuel, for always being ready for a cup of coffee. I also thank Erwann for the advice he shared with me at the beginning of my PhD. And, thank you, Cédric and Oliver for being my officemates for a couple of months.

I am very thankful to Iris for her huge help during my PhD journey, and all the nice chats we had at SCG. She made all the administrative stuff so easy!

I also thank David Röthlisberger and Romain Robbes for allowing me to visit and work for a month in Chile. It was an incredible experience!

One thank you to Clément Béra, for his collaboration and help regarding some of the work presented in this thesis.

I want to use this opportunity to thank my parents, who always believe in me and support me. I hope I made them proud. An enormous thank you to my husband for standing (by) me even when I was unbearable (who hasn't been like that at least once during PhD?!), and for always giving me courage and support.

I want to thank all of my friends from Čačak and Belgrade for always being there for me, and for helping home feel like home: Neda Niketić, Kaća Petrović, Ana Gutić, Isidora Čpajak, Jelena Petronijević, Jovana Pavlović, Mladen Pavlović, Jelena Ilić, Nina Radojičić, Ivana Milović, Stefan Mišković and Đorđe Bojović. Big thank you to all of my friends in Zürich for making Switzerland feel like home: Anja and Filip (and for all the board-games-playing weekends we had), Džimi and Alica, Marković and Irena. I am very thankful to my friends in Paris, Igor and Alina, for making my every trip to Paris enjoyable and for making me wish to be a student there again.

I thank all the people who directly, or indirectly, contributed to this thesis, and made it possible. Let the journey begin!

Audaces fortuna iuvat.

April 03, 2017
Nevena Lazarević

Abstract

Static type information facilitates program comprehension and analysis. Yet, such information is absent in dynamically-typed languages, and that increases the time needed for software maintenance. Type inference algorithms may provide type information to developers, but in order to be fast and assist in development phase, they sacrifice the precision for speed. One of the biggest obstacles for their precision is polymorphism presence.

In this thesis we first analyse the prevalence of polymorphism in object-oriented software, to assess the criticality it imposes on simple type inference. We find that polymorphism is omnipresent in object-oriented code, and that static analysis in dynamically-typed languages is also hampered by the usage of cross-hierarchy polymorphism, *i.e.,* duck typing. As this big obstacle for static code analysis cannot be bypassed, we propose the need for lightweight heuristics to tackle the problem of imprecision of simple type inference algorithms.

Four lightweight heuristics are employed to improve the performance of two simple and fast type inference approaches. These heuristics are founded on the source code and run-time information that are easy to collect without interrupting the workflow of a developer.

The heuristics are evaluated and compared with the underlying algorithms based on their inference time and precision. All of them show a significant improvement when compared to the basic algorithm. They introduce a negligible overhead on the inference time, thus we deem them usable during regular coding tasks.

Contents

1 Introduction

Due to the lack of static type information, developers in dynamically-typed languages need more time for software maintenance than developers in statically-typed languages [KHR+12, HKR+14]. Static type information is essential when it comes to understanding software control flow [SMDV08, KBR14], the core of the program comprehension process [RNDB09, SES05].

In an object-oriented language, regardless of whether it is statically- or dynamically-typed, the difficulty of control flow analysis depends, to a great extent, on the usage of inheritance and polymorphism in the code. Polymorphism enables genericity and extensibility by decoupling clients from providers, while inheritance allows code reuse. Yet, together they may cause the parts of the software that are semantically close to be scattered over various places in the code. If a class inherits or extends methods from its superclass, the intended behaviour of the class cannot be understood without understanding the behaviour of its superclass. It has been observed that a high use of inheritance poses serious complexity issues on program comprehension [Ous98].

This is particularly true in dynamically-typed languages where high use of polymorphism, combined with the lack of static type information, causes

an increase of navigation actions [RNDB09, KBR14], and consequently maintenance time. Let us consider the example in Listing 1. The example[1] is written in Smalltalk.

```
1 GLMLoggedObject subclass: #GLMPane
2    instanceVariableNames: '... presentations ...'
3    classVariableNames: ''
4    category: 'Glamour-Core'
5
6 GLMPane>>update
7    ...
8    self presentations do: [ :each | each update ]
```

Listing 1: The run-time type of the block argument `each` cannot be statically detected by the traditional approach

Lines 1-4 define a new class `GLMPane` which has an instance variable `presentations`, while the set of lines 6-8 define a method named `update`, responsible for updating each of the elements of the instance variable `presentations`. Let us suppose that the developer wants to understand how the update is performed. Smalltalk is a dynamically-typed language, thus there is no static type information. In order to navigate from a message send[2] `update` in the line 8 to the message implementation, the developer has at her disposal only the message selector, that is `update`. By using this traditional approach, the developer will obtain the list of seventeen[3] methods implementing the selector `update`, in thirteen different class hierarchies. In the situations like this, the developer is sometimes reluctant to explore all the method candidates statically, due to the possibly long list of selector implementors [KBR14].

The developer then usually attempts to acquire the type of the receiver and then seek the class implementing the corresponding message [KBR14]. If she fails, she is then forced to run the code to obtain the desired information, whereas static type information would eliminate the cost of running

[1] This code snippet is actual code from the Glamour-Core system: `http://www.smalltalkhub.com/#!/~Moose/Glamour/packages/Glamour-Core`

[2] In the rest of the thesis, we will mainly use Smalltalk terminology. In Smalltalk terminology, to invoke a service of an object, one "sends it a message". A message consists of a "selector" (the name of the message) and the arguments. The receiver is then free to decide which "method" that implements the corresponding "selector" to use to respond to that message.

[3] The system used for this example is Pharo 5.0, version 50761. The actual number of implementations may vary in the other systems.

the software. Besides, running the code just yields one possible software execution and hinders thorough understanding of the code [PTP07].

Even type information without explicit static type checking has a positive impact on development time [SH14]. However, wrong type information increases development time more than no type information [SH14]. It causes the developer to explore wrong parts of code, thus leads to the pollution of the working space and wasted development time [RND09]. Returning to the example in Listing 1, one would assume that the `presentation`-like objects would be stored in the instance variable `presentations`. This may lead the developer to explore the `GLMPresentation` class, which belongs to the same project as the code in the example, and its method `update`. However, the actual run-time type of an element of the `presentations` instance variable is a subclass of the `GLMPresentation` class. This subclass contains its own implementation of the `update` selector. We can conclude that the false type assumption would cause the wrong direction of the work flow, hence it would increase the development time. This emphasises the importance of *precision* for type information. Type information also needs to be obtained fast, in order not to disrupt the work flow [RNDB09, RND09, SH14, RL08]. This stresses the importance of *speed* for the underlying type inference.

For static type analysis to be precise, it must closely track control and data flow. These analyses provide the developer with the most reliable results possible, but they depend on the analysis of the whole program, making them slow and expensive. Modern software not only depends heavily on libraries, but is often a part of a distributed system, so the whole program may not even be available at analysis time. Furthermore, this kind of analysis has scalability issues [SS04].

Simple analyses statically track variable assignments and messages sent to a variable, and consequently pose constraints on the variable, the resolution of which would result in the set of possible types for the variable. Since they are neither control-flow nor data-flow sensitive, these approaches tend to be very fast. They are usable during coding tasks, since they would not break the developer's workflow. However, they tend to be less precise. Due to the trade-off of accuracy for speed, their precision is heavily hampered by polymorphism usage [Age95].

Yet little is known about the actual polymorphism presence in object-oriented software. While the observed difficulties in program comprehension in dynamically-typed languages [SMDV08, KBR14] are likely to be the direct consequence of the lack of static type information, they may also

suggest larger presence of polymorphic code in dynamically-typed than in statically-typed languages.

We therefore set out to investigate the actual usage of polymorphism in open source object-oriented software by studying two large corpora of open source software systems: one statically- and one dynamically-typed. The study revealed a high usage of polymorphism in both corpora, and a significant degree of scatteredness throughout the system. It provides us with two main conclusions: first — polymorphic code is omnipresent in both language corpora, and second — it is more present in dynamically-typed software, though the difference is not as large as are the differences in program comprehension in statically- and dynamically-typed software [SMDV08, KBR14]. Dynamically-typed code also makes a use of cross-hierarchy polymorphism, *i.e.,* usage of the same selector for methods defined in classes without a common superclass implementing that selector.

These findings suggest the importance of static type information for program comprehension. But, they also confirm the high presence of one of the biggest obstacles for static type inference. One of the fastest static type inference approaches, *RoelTyper* [PMW09], is directly hampered by the presence of (cross-hierarchy) polymorphism. The more the code is polymorphic, the more likely it is that this algorithm will *over-approximate* the set of possible types for a variable, that is, it will infer as possible types false positives, *i.e.,* classes that understand the interface of the variable (set of messages sent to the variable), but do not represent its run-time type.

We argue that *lightweight heuristics may be employed to improve the accuracy of control-flow and data-flow insensitive type inference algorithms,* that is, to mitigate the number of false positives, without a significant loss of speed. These heuristics are founded on the information that is easily accessible from the source code either statically or at run time. By performing lightweight code analysis, we believe that it is possible to augment the precision of a simple type inference algorithm, while preserving its simplicity and swiftness. Hence, they would remain fast and practical during coding tasks, without breaking work flow.

In order to illustrate this, we have implemented three of these heuristics on top of *RoelTyper* and one heuristic on top of the *Cartesian Product Algorithm* (abbreviated CPA) [Age95], which statically models type flow at run time. Our choice falls on these two algorithms, since they are fast enough to be usable during coding tasks. One more advantage is that they work with nominal types, as structural types burden program comprehension [Str].

The heuristics are based on static or run-time information. Proof-of-concept prototypes are implemented for Pharo[4], a dialect of Smalltalk, a highly reflective dynamically-typed language [GR83], that enables fast and easy implementation of analysis tools [FJ89]. We used these implementations to evaluate our claim. In both cases, we compared the results with the underlying algorithms, and demonstrated that the usage of heuristics increased the precision without significant overhead. To assess the criticality that polymorphism imposes on these simple algorithms, we measured the number of possible hierarchies to which types of the variables may belong. When using RoelTyper, more than 70% of variables have statically-inferred types that belong to two or more class hierarchies. This clearly indicates the degree of polymorphism usage throughout the code used for the purpose of evaluation. Implemented heuristics are able to correctly infer the types even for variables whose interface is understood by more than one hundred classes.

In the case of RoelTyper, we additionally compared the obtained results with *EATI (Ecosystem-aware type inference)*, an approach also built on top of RoelTyper [SLN14]. It uses the information from the language ecosystem to increase the precision of RoelTyper. While EATI needs more time and resources, it is less accurate than the implemented heuristics. We measured the time needed to provide a type feedback to assess whether these heuristics are fast enough to be used for program comprehension. The introduced overhead is less than 5% in case of RoelTyper and about 10% in case of CPA, which we deem acceptable.

1.1 Thesis statement

We formally state our thesis as follows:

Even in highly polymorphic code, the precision of simple type inference algorithms can be substantially improved by the use of lightweight heuristics established on easily accessible static and run-time information, while preserving the algorithm speed.

This thesis statement opens several research questions regarding the choice of type inference algorithms used as the basis, the choice of heuristics, the choice of the information from the source code used for the

[4] `http://www.pharo.org`

definition of the heuristics *etc.*

1.2 Contributions

In this section, we outline the main contributions of this thesis. The first contribution consists of a large-scale study of polymorphism in dynamically-typed languages, compared with its presence in statically-typed languages. The second contribution consists of the implementation of several heuristics on top of the existing simple type inference algorithms, and their evaluations. These contributions led to the publication of several scientific papers.

1.2.1 A large-scale software study on the prevalence of polymorphism in statically and dynamically-typed languages

The extent to which polymorphism is used in real programs, and its impact on program comprehension and development tools are not very well understood [DDM+03, DRW00, RHV+09, HLBAL05, WH92]. Even so, polymorphism represents one of the main obstacles for simple static analysis [PMW09, Age95, TP00]. To assess the criticality of polymorphism regarding program comprehension and static analysis, we investigated how prevalent its use is in object-oriented software. We analysed software systems written in Smalltalk, a representative of dynamically-typed languages, and in Java, one of the most widespread statically-typed languages. Since dynamically-typed languages pose more difficulties on developers than statically-typed languages in terms of program comprehension, we investigate whether the distribution of polymorphism is different in two language corpora. We have also analysed the extent of cross-hierarchy polymorphism usage in Smalltalk, also known as *duck typing*. The study shows that polymorphism is omnipresent in both language corpora and that it is more present in dynamically- than in statically-typed languages, though the difference is not large. Also, duck typing is used in almost all of the analysed dynamically-typed projects.

These findings were published in the proceedings of a technical track [MCL+15] and in the proceedings of an early research achievements track [MGN17b] of an international conference.

1.2.2 Lightweight heuristics for improving simple type inference algorithms

We present four heuristics developed to improve precision of simple type inference algorithms, by using easily retrievable information from the code. We implemented for each of the heuristics a proof-of-concept prototype, used for the evaluation. Each heuristic is based on a different kind of information that can be used for the improvement.

The simplest approach to infer types for a variable is to track down the assignments and set of messages sent to the variable [PMW09]. Even though this approach may be thought of as naive, it is very fast and precise for almost 60% of variables.

Since the approach does not track any flow of information throughout the code, one of its main obstacles is polymorphism usage. When the interface of a variable consists of popular selectors, implemented in multiple independent hierarchies, this approach offers ambiguous results. Since it concentrates on inferring a class hierarchy for the variable, rather than the concrete class, it may produce an excessive degree of false positives, *i.e.,* classes that understand the interface of the variable, but do not represent its run-time type. Each of the inferred types for a variable is more or less likely to represent its type at run time.

We do not want to change the design of an algorithm itself as it would increase its complexity, and the time needed for the analysis. That is why we propose ordering of the resulting classes. As each of the classes inferred as possible types for a variable is more or less likely to be correct, we explore possible heuristics for their sorting. Naturally, object that exists at run time needs to be created at some point during program execution. This can be achieved in any of the following ways:

1. invoking a constructor
2. invoking a (factory) method that plays the role of a constructor, but is not a constructor per se
3. via reflection

Our hypothesis is that the more frequently the class is instantiated in the code, the more likely it will represent a type of a variable at run time. Since this can be done in three different ways, we propose three heuristics implemented on top of RoelTyper.

On the other hand, in the presence of reflection, or dynamic class loading [HH09, RLBV10, RHBV11, CRTR13], static type inference algorithms miss certain types, hence they lose type information. Rather than over-approximating the set of possible types for a variable, they *under-approximate* it. This means that they suffer from the problem of false negatives, *i.e.,* they omit from the results the classes that represent a variable type at run time. We propose the fourth heuristic built on top of CPA, to improve its precision in the presence of reflection.

The following subsections present three heuristics for ordering classes, and the heuristic used to recover missed types, respectively.

Heuristic based on the class instantiation frequency

First, we propose ordering the inferred classes (or hierarchies) for a variable based on the frequency of constructor calls for a class. We have implemented a prototype and used it to evaluate the approach. The heuristic showed more than twofold improvement when compared with the basic approach.

The results obtained from the prototype implementation and evaluation have been published in the proceedings of an international conference [MN16].

Heuristic based on the class name occurrence frequency

While an instance of a class may be created by explicitly invoking a constructor of the desired class, it may also be created by the usage of a factory method [GHJV95]. Some languages do not pose restrictions on constructor naming [Bec97], thus it may be difficult to statically track all constructor invocations. Any method may play the role of a constructor.

We propose also to explore a heuristic of ordering possible types for a variable based on the frequency of class name occurrence in source code, rather than on the class instantiation frequency. We have implemented a proof-of-concept prototype, used to evaluate the heuristic.

Interestingly, this heuristic showed slightly better results than the previous one. We have compared it with EATI, an approach also built on top of the same basic algorithm [PMW09], which uses the information collected throughout the language ecosystem to order possible types for a variable. We show that with less effort, this heuristic outperforms EATI.

The results obtained from the proof-of-concept implementation have been published in the proceedings of an international conference, together with the results of the previously listed heuristic [MN16].

Heuristic based on the class frequency from inline caches

In some languages, like Smalltalk, Objective-C, Python and Ruby, classes can be used as first-class objects, *e.g.,* to receive a message or as an argument of a message send [BDN+09, PW88]. Recent studies show that reflective features are quite frequently used in dynamically-typed languages [HH09, RLBV10, RHBV11, CRTR13]. Due to dynamic class loading or high use of reflection, static analysis can miss the use of certain types [LSS+15]. This imposes difficulties for static analysis, as sometimes it cannot be known at compile-time which class will be instantiated or even created.

In the presence of dynamic binding, many virtual machines for dynamic languages employ Just-in-Time compilers to speed up the execution [DS84, HCU91a]. These compilers use inline caches, which locally store the information about methods previously executed for a message send. Beside method information, these caches also contain the information about the type of the receiver for a message send.

We hypothesise that the class usage frequency as a type of the receiver at run-time, read from the inline caches, may serve as a reliable proxy for the likelihood of a variable being of a certain type. This information can be used to order statically-inferred types. As run-time information is easily accessible from the virtual machine, no instrumentation is required.

We have implemented a proof-of-concept prototype, used to evaluate our hypothesis. The evaluation showed results very similar to those of the heuristics based on purely static information.

Results of this study have been published in the proceedings of an international workshop [MBGN16] and have been under review for a publication in an international journal.

Heuristic based on the type hints from method argument names

In the presence of reflection, or dynamic class loading, static type inference algorithms may under-approximate set of possible types for a variable.

On the other hand, in order to partially compensate for the lack of static type information, a common idiom in dynamically-typed languages is to

provide a type annotation for method arguments [Bec97, Zan13, Bol10]. These annotations are mainly intended to improve program comprehension, but they are also used as an input for some development tools, *e.g.,* code completion [BDN+09], in order to improve the results.

We hypothesise that these annotations from method argument names may be employed to improve the precision of a type inference in cases where the type of the variable cannot be statically inferred by traditional approaches. We propose a heuristic to augment a type inference algorithm whose precision significantly depends on the correctly inferred types for method arguments [Age95].

We have implemented a prototype used for evaluation. The obtained results show that the augmented algorithm outperforms the basic one significantly.

The results obtained from the prototype implementation and evaluation have been published in the proceedings of an international conference [MGN17a].

1.3 Outline

This dissertation is organised as follows:

Chapter 2 provides an overview of the related work with focus on different type inference techniques in dynamically-typed languages.

Chapter 3 presents the findings of the study of (cross-hierarchy) polymorphism presence in statically- and dynamically-typed languages and its possible impact on program comprehension and static analysis.

Chapter 4 presents two heuristics based on purely statically collected information about class usage and instantiation frequency, and provides the results of the evaluation.

Chapter 5 presents the heuristic based on the class usage information collected from inline caches and the corresponding evaluation.

Chapter 6 presents the heuristic based on the collected type hints from method argument names and its evaluation.

Chapter 7 concludes the thesis and discusses future work.

2 State of the art

In this chapter, we present the state of the art of current type inference techniques used in dynamically-typed languages. We start from the most exhaustive techniques and proceed towards simple ones. The chapter is divided into five sections, each of which covers a different approach of type inference. While the first two sections cover type inference techniques dependent on whole program analysis, the remaining sections are focused on partial type inference techniques.

2.1 Gradual typing

Gradual typing [ST06, ST07] represents a type system that can be enhanced over time. It allows some of the variables to be typed statically, while the rest of the software may be left untyped. Correctness of the typed part of the software is checked at compile time, thus the type system ensures that these types are respected at run time. If a run-time type error is raised, the corresponding untyped part of the software is indicated [WF07]. With gradual typing, a type error may originate only in the untyped part of the code or at the border of statically-typed and untyped code. If the complete

program is statically-typed, then no type error may arise at run time. Since gradual typing enforces type checks at compile time, it represents a halfway point between statically- and dynamically-typed languages.

Gradual typing has been implemented for Racket, a multi-paradigm programming language, which is originally untyped [THSA]. *Typed Racket* includes ways to type different language idioms, thus providing an easy transition from untyped Racket code. For example, if the predicate function `string?` is used for the purpose of dispatching, Typed Racket incorporates the corresponding type information in the two subsequent branches based on the provided answer, that is in the `then` branch the predicate returns `true` value, while in the `else` branch the returned value is `false`. Typed Racket then uses this information for the further analysis. In order to ensure soundness in the transition between untyped and typed part of the software, it installs contracts on the boundaries.

Typed Racket has been used as an inspiration for *Gradualtalk*, a gradual type system for Smalltalk, developed by Allende *et al.* [ACF+13]. Gradualtalk is based on extensive research of a large number of Smalltalk projects in order to accommodate a variety of programming features, like metaclasses and dynamic and reflective features. It includes nominal and structural types, as well as union types.

A gradual type system called *DRuby* [FAFH09, Fur09] has been implemented for Ruby, allowing developers to annotate and type check selected parts of the code base. Beside type annotation, it also employs type inference, in order to gradually provide a type for every object in the code. DRuby employs dynamic analysis for the sake of dynamic checking, that is to be able to check types in the presence of dynamic coding.

Typed Scheme offers a gradual typing for Scheme [TF11], similar to DRuby. It introduces occurrence typing, *i.e.,* different types assigned to distinct occurrences of the same variable, depending on the control flow. Like DRuby, it uses dynamic checks on the transition between untyped and typed code to ensure type soundness. Typed Scheme has been designed in a way to accommodate different language idioms and features, for example multiple value return option and the use of variable-arity functions.

Reticulated Python is a gradual type system for Python [VKSB14]. It includes structural types, as well as dynamic types and open types. Dynamic types have also been used in Gradualtalk [ACF+13] to accommodate any type that can be observed at run time. It provides an advantage over the `Object` type, since it does not require explicit casts in the code. Open types represent a form of union of two or more types, in a sense that

implicit downcasts are allowed.

Gradual typing has also been used in industry, via *ActionScript*, a gradual typing system for JavaScript [RCH12, Moo07].

All of these gradual type systems depend, in the first place, on the developer's type annotations, and then, gradually, may provide type information for the rest of the code, based on the type inference. Since gradual typing enforces static type checks, compiler will raise an error if type constraints are violated. This does not leave any space for imprecision, hence the employed type inference needs to be flow sensitive and, consequently, time-consuming. Also, if a part of the code is changed, the type inference needs to reanalyse the untyped part of the code from the start, in order to acquire precise information. Furthermore, we are interested in *fast* type inference techniques, intended for program comprehension, that do not need a whole program analysis in order to provide a type information, and that can be used in projects with no type information.

2.2 Optional typing

Optional typing grants to a developer the opportunity to specify the type of a variable when deemed necessary. While there is a static type checker in gradual typing, to ensure type correctness, optional typing does not guarantee the absence of type errors at run time [Bra04]. Even in the presence of a type error it would issue the report, but not prevent execution.

Three optional type systems have been developed for Smalltalk: Strongtalk [GJ90, BG93], Pegon [Smi] and TypePlug [HDN07]. Strongtalk supports structural types. It was used for optimisation purposes. Pegon represents an upgrade of Strongtalk, by performing type inference. TypePlug is an optional type system for Smalltalk that provides the possibility to insert type annotations and enables type checking on demand. Gradualtalk may also serve as an optional type system, in which case, run-time checking is deactivated.

Groovy offers the possibility of optional typing without static type checking [Sub13]. A recent study revealed the way developers use optional typing in Groovy [SF14]. Developers with less background in dynamically-typed languages tend to use type annotations more often, mostly in method definitions. Furthermore, types are less used in frequently changed code.

The Dart[1] programming language also offers an optional type checker

[1] `http://www.dartlang.org/`

that can be enabled on demand. An empirical experiment performed with Dart developers showed that type annotations serve as a kind of documentation [FNT15].

TypeScript is an optional type checker for JavaScript [BAT14] that aims to host different language idioms, like covariance of a property, and enables an easy transition from untyped JavaScript code.

Vitousek *et al.* present a way to transition from optional to gradual typing in Python by inserting checks to ensure that there is no type error [VKSB14]. This approach was then also applied to TypeScript [VS16].

Maidl *et al.* present an optional type system for the scripting language Lua [MMI14], which offers some uncommon features like functions with unfixed number of arguments.

A pluggable type system allows a language to support multiple, optional type systems [Bra04, ANMM06]. Similar to gradual typing, the main drawback of an optional type system from our research perspective is that the corresponding type inference is focused on complex whole-program analysis, whereas we aim for fast type inference algorithms that do not depend on the developer's input.

2.3 Static type inference

The pioneer in this field was Milner [Mil78], who developed a type inference algorithm for ML, a functional programming language. The algorithm supports parametric polymorphism, but not subtyping. It infers the type of a variable using constraints based on the usage of a variable. The algorithm is sound: if an ML program is well-typed, it will not produce run-time type errors. Since it does not take subtyping into consideration, it is of limited interest to us.

Another type inference algorithm is proposed for the functional language FL [AM91]. FL is a dynamically-typed language, in which a type is considered to be a set of normal-form expressions. Instead of computing types as sets of values, they are computed as sets of expressions, thus using an operational view of expressions. One of the main issues is performance. However, in this thesis we concentrate on the inference of nominal types, since structural types burden developer's reasoning [Str].

Inferring types of an expression from the set of constraints imposed on the expression was performed by Palsberg *et al.*. It was first developed for a language similar to Smalltalk, but which avoids some of the commonly

used Smalltalk features [PS91]. The algorithm is named the *Basic Algorithm*. It was later implemented for the *Self* programming language [US87] by Agesen *et al.* [APS93]. Self is based on cloning objects rather than instantiating classes, and it offers a support for dynamic and multiple inheritance. This is the first algorithm to consider dynamic and multiple inheritance.

Agesen continued his work on type inference for the Self programming language. In order to optimise his previous type inference algorithm for Self, he developed the *Cartesian Product Algorithm* [Age95]. The Cartesian Product Algorithm, known as CPA, exploits the fact that the type of the return value of a method usually depends on the types of the method arguments. Thus, whenever a new message send is to be analysed, the algorithm creates the Cartesian product of all of the inferred types of the method arguments, including the receiver types, and analyses each of the tuples separately. Thus, it provides more precise information for a method's return type than the Basic Algorithm.

Starkiller [Sal04] is a type inference algorithm for Python, based on the Cartesian Product Algorithm. It tries to reconstruct a flow of types in order to model the run-time behaviour of a software. Each variable is represented by a node that holds information about possible types that a variable may assume at run time.

Ecstatic [MSK07] is an implementation of type inference for Ruby programming language based on CPA, adapted for various Ruby programming idioms, for example block usage, as there are seven distinct block syntaxes.

CPA was initially developed for Self, a prototype-based programming language, thus its implementation in class-based languages is hampered by polymorphism usage [Age95]. This issue was addressed in DCPA [WS01], an implementation of CPA for Java that deals with data polymorphism. This implementation is used to check whether downcasts in Java are correct. It showed a significant improvement in precision when compared to CPA, while the efficiency remained on the same level.

The precision of CPA heavily depends on the correctly inferred types for method arguments. As most type inference algorithms lose their precision in the presence of reflection and dynamic coding [LSS+15], commonly used features in OO languages [CRTR13, BSS+11, RLBV10], it is also the case with CPA. We consider that this algorithm would benefit the most from type annotations in method argument names used in dynamically-typed languages [Bec97, Zan13, Bol10]. Since CPA is very

fast, thus usable for program comprehension, we choose it as the basic algorithm on top of which we implement the heuristic based on type hints from method argument names.

Spoon *et al.* have developed *Chuck*, one of the most precise type inference algorithms [SS04, SS05]. It is a demand-driven algorithm that uses subgoal pruning. The information used for type inferencing is analysed on demand, and the precision of the algorithm decreases if some of the subgoals required for type inference need to be discarded for complexity reasons. The main problems are scalability and performance, thus its usage during development is questionable.

JS_0 represents a type inference algorithm for JavaScript [AGD05] with the support for dynamic features like field or method additions at run time. The algorithm uses the notion of structural types. Our focus is on nominal types.

One of the simplest but fastest approaches for static type inference, the *RoelTyper* algorithm, was developed by Wuyts *et al.* [PMW09]. Even though the algorithm may be considered naive, since type inference is based only on statically tracking down the set of messages sent to a variable and on assignments to it, it has shown promising results. However, for a significant portion of variables, the offered results are ambiguous. Since the approach is not flow-sensitive in any way, *i.e.,* it is neither control- nor data-flow sensitive, its precision suffers in the presence of polymorphic and duck-typed code. Since the approach is mainly intended for program comprehension, it is acceptable, but due to the false positives in its results it may lead to wasted developer's effort [KBR14].

RoelTyper is a very fast type inference algorithm, with space for possible improvement in the presence of polymorphic and duck-typed code, thus we opt for its improvement in Chapters 4 and 5.

This approach was also used as a basis for the prototype implementation of the EATI (Ecosystem-aware type inference) algorithm, which advocates the idea of using the information available in the language ecosystem to increase the precision [SLN14]. It statically tracks the frequency of messages being sent to the instances of different classes in the language ecosystem. Based on these frequencies and the set of messages sent to a variable, it calculates the likelihood of the variable being of a certain type. EATI has shown almost 100% improvement when compared with RoelTyper.

EATI offers a way to improve the precision of a simple type inference technique. We used it as an inspiration for the heuristics implemented on

top of RoelTyper. However, since it depends on the data collected from the ecosystem and queried from a central repository, in our experience, it takes a more time than the implemented heuristics to compute type information. We strived to acquire a faster and more precise approach.

2.4 Dynamic type inference

Beside using statically collected information, type inference approaches may also employ information collected at run time, *i.e.,* from a program execution. This information can be collected either through code instrumentation [LB94] or from the virtual machine directly [HCU91a, HU94]. We divide this section into two parts, accordingly.

2.4.1 Code instrumentation

Rubydust is a constraint-based type inference algorithm for Ruby, dependent on source code instrumentation and dynamic execution of the program, developed by An *et al.* [ACFH11]. The approach uses as input types recorded during the execution of the program for which the types need to be inferred. Rubydust observes the usage of variables, for example as message receivers, and subsequently generates subtyping constraints based on the variable types observed at run time. In order for the analysis to be sound, the program execution must cover all possible paths in the control-flow graph. Hence, as the authors state, the current execution overhead introduced with this analysis is quite high. Type inference information is, afterwards, introduced statically, based on the information collected during program runs.

Type inference that relies on dynamic collection of data has been developed for Smalltalk [RBFDD98]. The algorithm observes types of objects at run time and incrementally updates static type information. The main assumption made for this work is that test coverage is "complete" and that the program of interest is in a runnable state. Obviously, the more frequently the variable has been encountered during a program run, the more precise the type information will be.

JavaScript is of interest when it comes to type inference based on types observed at run time. Odgaard *et al.* present a way of annotating JavaScript code based on the run-time type information collected from running unit tests [Odg14]. The two main problems in this work are memory usage and

performance.

Pradel *et al.* implemented *TypeDevil*, a tool that informs developer about type inconsistency at run time [PSS15]. TypeDevil records types of variables and functions at run time and informs a developer if the corresponding types are inconsistent.

From our research perspective, dynamic type inference based on code instrumentation has two drawbacks: execution overhead and the need to run the code. Some dynamically-typed languages, *e.g.,* Smalltalk, do not have a fixed entry point (like, for example, the *main* method in Java), thus any method in the project can be used as starting point for the execution. Having to run the code in order to obtain type information can be a burden for a developer that we aim to bypass. It forces the developer to redirect her focus from the analysed code to something else and break her work flow. Furthermore, due to the execution overhead, it is not convenient for use in time-sensitive applications.

2.4.2 Inline caches

Hölzle *et al.* were the first to advocate the usage of polymorphic inline caches for the sake of type feedback [HCU91b]. One of the first uses of inline caching as a way to statically reconstruct the type of a variable from a running system was in Self [HU94]. The authors collect run-time receiver types observed at each message send, and feed this information back to the compiler for optimization purposes. They predict the type of the receiver based on the receiver's types observed during previous program runs.

Types collected from inline caches, *i.e., type feedback* has been used for optimisation purposes in C++ programs [AH96]. The authors combined it together with *Class Hierarchy Analysis* [DGC95, Fer95] to reduce the number of virtual function calls. Class Hierarchy Analysis is an algorithm used to construct a call graph from a program. It creates a set of method candidates for a function call based on the statically-declared type of the receiver. The algorithm reduced significantly the number of virtual function calls and it improved the performance, *i.e.,* decreased the execution time by 40% on average.

Agesen *et al.* preformed a comparison between type feedback, *i.e.,* obtaining types from the program execution, and type inference, as performed by the Cartesian product algorithm [AH95] on a set of twenty three Self programs. In their benchmarks, both techniques inlined about 95% of virtual call sites.

While running instrumented code mostly introduces significant overhead in program execution, this is not the case when it comes to the information collected directly from the virtual machine. As for code instrumentation, the main drawback from our perspective is that this kind of type feedback requires program execution, which may burden the developer. However, since it does not introduce execution overhead, we used inline caches in order to collect type information over time (and not at the moment of type inference). We then employ it on top of the static type inference algorithm. This way we do not burden the developer with the need to run the project herself.

2.5 Other techniques

The combination of static and run-time analysis is known as *hybrid analysis* [Fla06, KF10]. The key idea in this approach is to infer conservative information by the use of static analysis, which is further fine-tuned by information collected at run time. These approaches are useful in cases where static analysis tends to produce too conservative data, or where it is unsound due to dynamic class loading or reflection.

Fast and precise hybrid type inference has been presented for JavaScript [HG12]. It tries to infer sound type information by customising static type information to also deal with the types recorded at run time.

Soft typing also represents one form of the combination of static and dynamic typing [CF91], implemented for Scheme. Different from hybrid analysis, this approach can statically check the program and insert run-time checks for the variables for which it cannot surely infer the type. It was later implemented for Erlang [Nys03].

3

Study of polymorphism prevalence

3.1 Introduction

Polymorphism in programming languages indicates the possibility of using the same variable to holds values of different types. In object-oriented languages, subtype polymorphism (also known as *data* polymorphism) means that the set of messages that can be sent to a variable is determined based on its (declared) type, while at run time the variable may point to an instance of a subtype of its (declared) type. Since there is no static type declaration in dynamically-typed languages, a variable can hold instances of any type at run time (preferably understanding the messages sent to that variable), even if these types are unrelated.

Polymorphism represents the essence of object-oriented coding style. Even though it allows developers to write easily extensible software [MH90, DRW00], it also burdens program comprehension [DDM+03, DRW00, KBR14]. A particular selector may be implemented in related classes that are scattered throughout the code. Since a class may inherit a complete method from a superclass or may extend it, the intended behaviour of the class cannot be understood without analysing the corresponding method of the superclass.

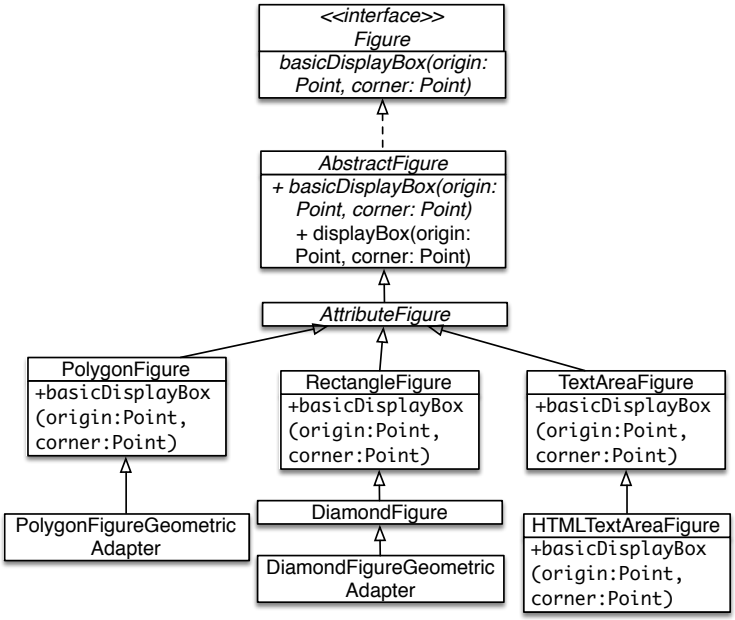

Figure 3.1: Sample class hierarchy from JHotDraw, with multiple implementations of the operation `basicDisplayBox(Point, Point)`.

To better understand the impact polymorphism has on program comprehension and analysis, let us consider the classes in Figure 3.1 from JHotDraw[1], a Java framework for editing structured graphics. The class diagram shows a subset of the JHotDraw `Figure` hierarchy[2]. A developer is trying to understand the behaviour of `AbstractFigure`, one of the key classes of JHotDraw.

The snippet in Listing 2 shows an instance of the template method design pattern. The receiver of the message send `basicDisplayBox(Point, Point)` (line 6, Listing 2) is the implicit variable **`this`**, which can be bound to any subtype of `AbstractFigure`.

[1] JHotDraw is a reimplementation by Eric Gamma of HotDraw, originally developed by John Brant in VisualWorks Smalltalk.

[2] The complete hierarchy under `AbstractFigure` contains 35 classes.

```
public abstract class AbstractFigure
    implements Figure {
 //...
 public void displayBox(Point origin,Point corner){
 willChange();
 basicDisplayBox(origin, corner);
 changed();
 }
 //...
}
```

Listing 2: A polymorphic message send

Since the method `basicDisplayBox(Point, Point)` in the `AbstractFigure` class is abstract and any subtype of `AbstractFigure` can provide its own implementation, the method that will be actually invoked cannot be statically determined.

The developer could set a breakpoint in the code and observe which of the multiple implementations are invoked at run time, but this usually gives just a narrow selection of all possible invocations [PTP07], and introduces the costs of running the software [Ous98].

In situations like this, in dynamically-typed languages developers sometimes resist using the IDE navigation tools, because they suspect that the results would be polluted by faulty data [RHV+11], due to the lack of static type information [KBR14]. Although object-oriented systems highly depend on the polymorphism [RHV+09], very simple type inference algorithms that may be used as input for these tools, are able to produce unambiguous results for a bit less than 60% of variables [PMW09]. The information for the rest of the variables is far too broad to be used for program comprehension, *i.e.,* contains false positives. This is an indicator of a significant number of selectors implemented in unrelated classes, *i.e.,* in classes which do not have a common superclass implementing the same selector. It is obvious that polymorphic code would greatly benefit from type information, yet it represents one of the biggest obstacles for static type analysis.

We continue by defining terms that will be used in the rest of the chapter. We call the number of methods that could potentially be invoked by message send at run time the *cardinality* of the message send. With a slight abuse of terminology, we consider a message send to be *polymorphic*

if its cardinality is greater than one, a selector polymorphic if it is the selector of a polymorphic message send, and a method polymorphic if it implements a polymorphic selector (see Section 3.3).

Returning to the example in Figure 3.1, the cardinality of the polymorphic message send `willChange()` equals just two, while the cardinality of the message send `basicDisplayBox(origin, corner)` is 18. A higher cardinality leads to more behaviour being scattered through the system, and this will likely impact program understanding [DDM+03, DRW00, HLBAL05, WH92].

We investigate in this chapter the prevalence of polymorphism in object-oriented software. To the best of our knowledge, there is no large-scale study on the prevalence of polymorphism in open source software. We therefore set out to investigate the actual usage of polymorphism in open source software by studying two large corpora of open source software systems, and by posing the following research questions:

RQ1) How prevalent are polymorphic methods in object-oriented systems?

RQ2) How common are polymorphic message sends?

RQ3) What is the distribution of the cardinality of polymorphic message sends?

For the purpose of this study we consider only *static* information, that is, we do not consider how much polymorphism actually occurs at run time. Although this will only give us an upper bound on the actual polymorphism presence, we argue that this provides a good estimate of the challenges faced by a programmer reading the source code and obstacles that polymorphism imposes on static type analysis.

Answering the first question will reveal how big is the part of the software involved in defining polymorphic selectors, *i.e.,* how big is the part of software which possibly affects the ambiguity in program comprehension and analysis. The response to the second question will estimate how likely it is to encounter a polymorphic message send, and with the answer to the third question we can estimate the difficulty of the analysis of a polymorphic message send.

Nevertheless, program comprehension and analysis in dynamically-typed languages is further hampered by cross-hierarchy polymorphism, *i.e., duck typing* [TFH09]. The name refers to the "duck test", by James

Whitcomb Riley: “When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck”. Duck typing refers to the usage of a variable to point to the objects of distinct classes that understand the same set of messages, without a common superclass understanding those messages. A classic example of duck typing in Smalltalk is the interchangeable usage of `Symbol` and `BlockClosure` classes, since instances of both classes understand the message `value:`. In other words, a `Symbol` object can behave like a `BlockClosure` object (*i.e.,* it may “quack like a duck”) by responding to the `value:` message. These variables usually demand flow-sensitive algorithms for their precise inference. Duck typing can be mimicked also in statically-typed languages, with the use of interfaces. Yet, the statically-declared type of a variable explicitly states and restricts the variable's type to a certain interface, preventing it to be a subtype of any other interface declaring the same selector. This information is omitted in dynamically-typed languages and the variable may hold an instance of *any* available class that, supposedly, understands the interface of the variable. Therefore, in addition to the first three research questions, we address one more concern in dynamically-typed languages: the prevalence of duck typing. We say that a method is *duck-typed* if it has the same selector as another method, neither of which overrides a method with the same selector of a common parent. Consequently, the method selector is considered to be *duck-typed*, and a message send is *duck-typed* if its selector is duck-typed. Hence, we ask the following research questions:

RQ4) How prevalent are *duck-typed* methods in dynamically-typed software?

RQ5) How prevalent are *duck-typed* message sends in dynamically-typed software?

The rest of this chapter is organised as follows: we discuss the related work in the Section 3.2, then define our terminology in Section 3.3. Next we introduce our experimental methodology and the analysis infrastructure in Section 3.4. In Section 3.5 we report on the findings regarding the prevalence of polymorphism in practice. We then describe potential threats to the validity in Section 3.6. In Section 3.7 we discuss the implications our results entail and propose a series of questions to pursue in the follow-up study before concluding in Section 3.8.

3.2 Related Work

The influence of the depth of inheritance on code maintenance has been researched by Daly *et al.* [DBM+96]. They found that a system with three levels of inheritance was easier to maintain than a corresponding system with no inheritance. Yet, a system with five levels was found to take longer to modify than the corresponding system with up to three levels of inheritance. These results were contradicted by Cartwright *et al.* and Harrison *et al.*, who replicated Daly *et al.*'s study [Car98, HCN00]. Cartwright *et al.* found inheritance to have a positive effect on maintenance time [Car98], whereas the study of Harrison *et al.* revealed that a system with zero inheritance was easier to modify than the equivalent systems with three or five levels of inheritance [HCN00].

Tempero *et al.* focused on the presence of inheritance in Java code, rather than on its influence of code maintainability [TNM08]. They analysed the code in 97 Java systems. Their work showed a high use of inheritance, differences in the use of inheritance between interfaces and classes, and a different use of inheritance when applied to external libraries. While in their work the aim was the study of inheritance in general, we focus on the investigation of polymorphic methods in the context of inheritance hierarchies.

A similar empirical study in the terms of the size of analysed software was performed by Grechanik *et al.* [GMD+10]. Their study focused on the analysis of common source code patterns in a large-scale open source code repository, composed of 2080 Java projects from Sourceforge. They discovered that most inheritance hierarchies are flat, *i.e.,* have a depth of one and that almost half of the classes do not inherit from any superclasses, but just the implicit class `Object`. They also discovered that few methods are overridden, and that most methods have zero or one argument. Even though they analysed various aspects of object-oriented development, they did not focus their attention on polymorphism presence in OO code, as we do.

First study about method overriding — a concept closely related to polymorphism — was performed by Briand *et al.* [BWDP00]. They analysed how fault-prone is the code containing overriden methods in C++ code.

Tempero *et al.* conducted an empirical study of method overriding in a corpus of 100 open source Java systems [TAD+10]. Even though they analysed various aspects of method overriding, *e.g.,* the number of overriding

methods, the number of inherited methods, and the number of classes with replaced implementations, their focus was not on polymorphism usage. They did not focus on the call site analysis, as we do in this study. Their study showed that most subclasses override at least one method and many classes only declare overriding methods.

The usage of parametric polymorphism was studied by Parnin *et al.* in Java code [PBMH11]. They found that it was used by developers after its release, though mainly by one developer among all contributors to the code. Also, the old code was not often converted to use generics. Our study differs from theirs as we analyse subtype polymorphism only, as parametric polymorphism is more concerned with genericity. In addition, Smalltalk does not support parametric polymorphism.

Duck typing has been analysed on a set of 36 Python programs [ÅW15]. Based on the recorded run-time types of variables, the authors state that most variables are monomorphic, *i.e.,* point to objects of only one type at run time. However, most of the rest of the variables do point to the objects of unrelated types, *i.e.,* duck typing is used at run time. We are not aware of any large-scale study concerning the prevalence of duck typing.

3.3 Terminology

In this section we make precise the notion of subtype polymorphism and duck typing. To this end, we introduce the following simple set-theoretic model summarised by the UML diagram in Figure 3.2.

$$\mathit{msg} : \mathit{MS} \to S \quad (3.1) \qquad \mathit{sup} : C \to C \quad (3.5)$$

$$\mathit{def}_{ms} : \mathit{MS} \to M \quad (3.2) \qquad \mathit{rec} : \mathit{MS} \to T \quad (3.6)$$

$$\mathit{sel} : M \to S \quad (3.3) \qquad \mathit{impl} : T \to \mathcal{P}(I) \quad (3.7)$$

$$\mathit{def}_{m} : M \to C \quad (3.4) \qquad \mathit{sel}_{t} : T \to \mathcal{P}(S) \quad (3.8)$$

Given all source code of a system, C is the set of all classes, I is the set of all interfaces, T the set of all types, M the set of all methods. S is the set of all selectors, *i.e.,* method names in Smalltalk (since we do not

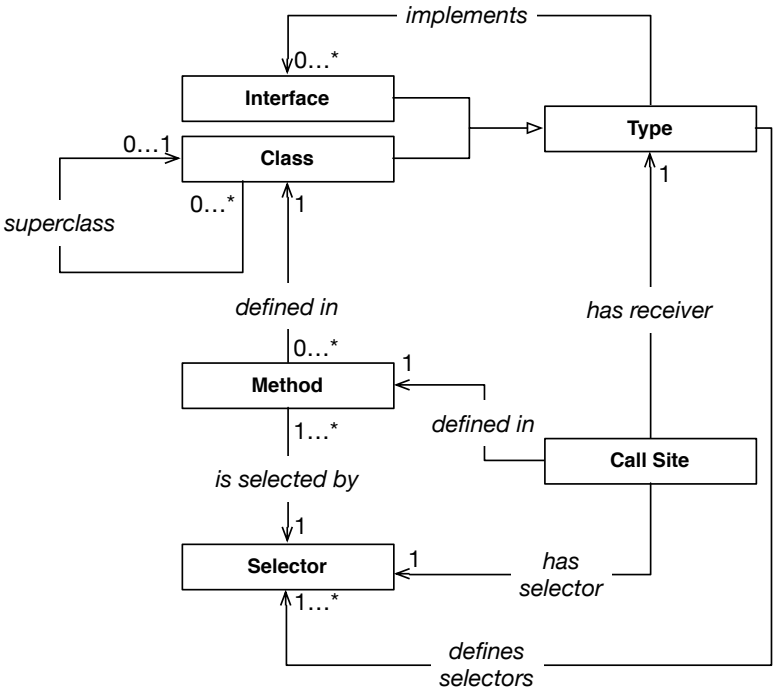

Figure 3.2: The core model in UML. The entities Interface and Type are relevant for Java, but not Smalltalk.

know the static type of method parameters) and method signatures in Java. *MS* is the set of all message sends[3] in the system.

Each message send ms has a selector $s = msg(ms)$ (3.1) and is defined in a unique method $m = def_{ms}(ms)$ (3.2). We say that the method m defines the message send ms and that method m sends the message s. Each method m has a unique selector $s = sel(m)$ (3.3), and is defined in a unique class $c = def_m(m)$ (3.4). Class c either has a unique superclass

[3]Message send refers to a call site in Java terminology.

$c' = \mathit{sup}(c)$ (3.5) or does not have a superclass.[4] We denote by c^* the *superclass-chain* of the class c (3.9), and we consider $\mathit{sup}^0(c) = c$ (3.10). A set of classes $H \subset C$ is a hierarchy if every two classes $c_1, c_2 \in H$ have at least one common class in their superclass-chains, the intersection of their superclass-chains is also contained in H, and for every class $c \in H$ there is no "gap" in its superclass-chain within H (3.11).

$$c^* = \mathit{sup}^*(c),\ \mathit{sup}^*(c) = \bigcup_{k \geq 0} \mathit{sup}^k(c) \tag{3.9}$$

$$\mathit{sup}^k(c) = \mathit{sup}(\mathit{sup}^{k-1}(c)),\ k \in \mathcal{N} \tag{3.10}$$

$$\begin{gathered}(\forall c_1, c_2 \in H)((c_1^* \cap c_2^* \neq \emptyset) \wedge (c_1^* \cap c_2^* \subset H) \\ \wedge \\ (\forall c \in H)(\nexists k, n \in \mathbb{N}_0)(n > k)(\mathit{sup}^n(c) \in H \wedge \mathit{sup}^k(c) \notin H))\end{gathered} \tag{3.11}$$

In Java, each message send ms has the static type of the receiver $t = \mathit{rec}(ms)$ (3.6). The type of the receiver can be either an interface or a class. Each type t has a set of interfaces it implements $i_t = \mathit{impl}(t)$[5] (3.7), and a set of selectors it defines $s_t = \mathit{sel}_t(t)$ (3.8). This information is not available for Smalltalk, and is not modeled.

Consider the example in Listing 2. For the message send in line 6, $ms =$ `basicDisplayBox(origin, corner)` the selector (a Java signature) is $\mathit{msg}(ms)$ = `basicDisplayBox(Point, Point)`, the receiver type is $\mathit{rec}(ms)$ = `AbstractFigure`, and the message send is defined in the method $\mathit{def}_{ms}(ms)$ = `AbstractFigure>>displayBox(Point, Point)`.

We can now query the model to compute the metrics necessary to answer our research questions, as summarised in Figure 3.3.

Definition 1. *A message send* ms *is* polymorphic *if there is more than one method that can be invoked at* ms *at run time.*

We determine this in slightly different ways in Smalltalk and Java, since we lack static type information in Smalltalk. In order to find all

[4] *sup* is a partial function, since we consider only classes defined locally in the corresponding project (*i.e.,* ignoring classes from external frameworks or the base system — *e.g.,* class *Object*).

[5] To avoid the usage of another function, we say that an interface implements another interface.

$$
\begin{aligned}
\mathit{subh}(t) &= \begin{cases} (\mathit{sup}^{-1})^*(t) \cup t, \text{if } t \in C \\ \{(\mathit{sup}^{-1})^*(t'), t' \in\ t^* \cap C\} \cup t^* \\ \quad \text{where } t^* = t \cup (\mathit{impl}^{-1})^*(t), \text{if } t \in I \end{cases} && (3.12)\\
\mathit{impl}(t,s) &= \{m \in M |\ \mathit{sel}(m) = s \wedge \\
&\qquad \mathit{def}_m(m) \in \mathit{subh}(t)\} && (3.13)
\end{aligned}
$$

Smalltalk:

$$
\begin{aligned}
\mathit{undr}(c,s) &= s \in \mathit{sel}_t(c) \vee \mathit{undr}(\mathit{sup}(c), s) && (3.14)\\
\mathit{intr}(s) &= \{m \in M |\ \mathit{sel}(m) = s\ \wedge \\
&\qquad \neg \mathit{undr}(\mathit{sup}(\mathit{def}_m(m)), s)\} && (3.15)\\
\mathit{isp}(ms) &= (\exists m \in \mathit{intr}(\mathit{msg}(ms)))\ s.t. \\
&\qquad (|\mathit{impl}(\mathit{def}_m(m), \mathit{msg}(ms))| > 1) && (3.16)\\
\mathit{isp}(s) &= (\exists m \in \mathit{intr}(s))\ s.t. \\
&\qquad (|\mathit{impl}(\mathit{def}_m(m), s)| > 1) && (3.17)\\
\mathit{duck\text{-}typed}(s) &= |\mathit{intr}(s)| > 1 && (3.18)\\
\mathit{duck\text{-}typed}(ms) &= |\mathit{intr}(\mathit{msg}(ms))| > 1 && (3.19)
\end{aligned}
$$

Java:

$$
\begin{aligned}
\mathit{isp}_t(ms) &= |\mathit{impl}(t, \mathit{msg}(ms))| > 1 && (3.20)\\
\mathit{isp}_t(s) &= (\exists t \in T)\ s.t.\ (|\mathit{impl}(t,s)| > 1) && (3.21)
\end{aligned}
$$

Figure 3.3: Computing polymorphism metrics

polymorphic message sends we first introduce the function that maps a type t to its complete subhierarchy (3.12).

Consider the example in Figure 3.1. For t = `RectangleFigure` subhierarchy is the set of classes

$\mathit{subh}(t)$ = {`RectangleFigure`, `DiamondFigure`, `DiamondFigureGeometricAdapter`}

while for t = `AttributeFigure`

$\mathit{subh}(t)$ = {`AttributeFigure`, `PolygonFigure`, `PolygonFigureGeometricAdapter`, `RectangleFigure`, `DiamondFigure`, `DiamondFigureGeometricAdapter`, `TextAreaFigure`, `HTMLTextAreaFigure`}

We also introduce the function *impl*(t, s) which yields the set of methods implementing the selector s and defined in the subhierarchy of the type t (3.13).

In Smalltalk we do not know the type of the receiver at compile time, so we must investigate each message send based only on its selector. To do this, we use the function *undr*(c, s) to determine whether the class c understands the selector s, either because it defines the method m such that *sel*$(m) = s$ or because one of the classes in its superclass-chain does (3.14). We also use the function *intr*(s) to find all the methods *introducing* the selector s, *i.e.,* such that *sel*$(m) = s$ and being defined in the class that does not have the superclass which understands the selector s (3.15). We then say that the message send ms is polymorphic, written $isp(ms)$, if there exists a method m introducing the selector *msg*(ms) and having at least one more method with the selector s in the subhierarchy of its class *def*$_m(m)$ (3.16). In Smalltalk we do not consider as polymorphic those methods implementing the same selector, but defined in classes without a common superclass also defining that selector, *i.e.,* duck-typed methods. These methods are explored separately.

In Java, for each message send ms we know the compile-time type of the receiver, $t =$ *rec*(ms), so we need to define the polymorphic message send with respect to t. We say that the message send ms is polymorphic if there are at least two methods implementing the selector *msg*(ms) and being defined in the subhierarchy of the type t (3.20), regardless whether t is an interface or a class. Our decision to treat a message send the same in both cases and possible threats to validity are discussed in the Section 3.6.

Definition 2. *A selector* s *is* polymorphic *if it can be a selector of a polymorphic message send. By extension, we consider a* polymorphic method *to be a method that implements a polymorphic selector.*

Again, since in Smalltalk we do not have the information about the static type of the receiver, this means that the selector s is polymorphic if there is at least one class c defining a method m such that *sel*$(m) = s$ and having at least one class in its subhierarchy defining another method with the same selector (3.17). In Java, we define the term of polymorphic selector with respect to the possible type of the receiver (3.21).

Consider the example in Listing 2. The selector $s =$ `basicDisplayBox (Point, Point)` is not polymorphic with respect to the possible type of the receiver $t \in$ {`HTMLTextAreaFigure, DiamondFigure`}, but *isp*$_t(s)$ = *true*,

for $t \in$ {`TextAreaFigure, AbstractFigure`}.

To assess the criticality that potential duck typing imposes on code analysis and understanding in dynamically-typed languages, we explore the methods that have the same selector, but defined in classes without a common superclass defining the same selector, and messages sends having these selectors.

Definition 3. *A selector* s *is* duck-typed *if there are at least two methods in the system introducing that selector* s*.*

This definition is modelled by Equation 3.18. Consequently, elements of the set *intr*(s) are *duck-typed* methods.

Definition 4. *A message send* ms *is* duck-typed *if its selector msg*(ms) *is duck-typed.*

This definition is modelled with the Equation 3.19. The *degree* of message send ms with regard to duck typing is equal to the number of methods introducing the selector of ms, *i.e.,* $|$*intr*(*msg*(ms))$|$.

3.4 Experimental Setup

Our study covers 111 systems written in Java and 1,128 systems written in Smalltalk.

We chose Smalltalk for its reputation as a "pure" object-oriented language (Smalltalk goes as far as implementing conditionals as polymorphic methods in the `Boolean` class hierarchy), and Java as a representative of a widely used, pragmatic object-oriented language. We statically analyse these systems to gather information about the usage of polymorphism and complexity it imposes on code analysis.

- For the Smalltalk part, we took a snapshot of all the 1'850 software projects stored in the SqueakSource repository in early 2010. At that time SqueakSource contained the majority of all projects implemented in the open-source Smalltalk dialects Squeak and Pharo and hence provided a representative set of Smalltalk projects from both industry and academia. We limit our analysis to projects containing more than 50 classes in order to exclude student projects and other small and likely less relevant projects. Out of the 1'850 projects, 1'128 projects meet this criterion. These projects contain 125'825 classes and 1'637'228 methods in total.

- For Java we selected 111 open-source projects from the Qualitas Corpus — a curated collection of software systems representing widely known open-source Java software systems and libraries [TAD+10]. Although we suspect so, we cannot guarantee that the selection is representative of well-engineered and maintained open source Java software. The corpus consists of over 130'000 classes and 1'086'000 methods. For the Java corpus, we use the Pangea analysis infrastructure [CCSL14] which enables us to easily deploy our analysis on the entire Qualitas corpus.

3.4.1 Data processing

Each project is parsed in order to extract the relevant metrics. We employed Ecco and Monticello as parsers for the Smalltalk corpus [RLR12], and VerveineJ and Moose for the Java corpus [DGN05]. The data processing consisted of two steps:

1. **Method analysis.** To measure how polymorphism is used we traverse the body of every class in the system, each class c having a list of methods $def_m^{-1}(c)$ it implements. We keep track of the set of methods implemented in the project, as well as of the set of all methods that are either overridden or are overriding, and the set of their selectors. These are polymorphic selectors. We also store the information about methods that introduce a selector. We then calculate the metrics related to polymorphism and duck typing, such as which methods in a system are involved in polymorphic and duck-typed message sends.

2. **Message send analysis.** In the second step, we traverse the body of each method, detecting all message sends within the method body. We then collect separately all the message sends ms for which $isp(ms) = true$ and all the message sends for which $duck\text{-}typed(ms) = true$, as well as their cardinality and degree, respectively.

3.4.2 Data analysis

We are primarily concerned with the distribution of the metrics over the corpus, and, as a secondary concern regarding polymorphism, whether they are distributed similarly in Smalltalk and in Java. To this aim, we

use box-plots to summarise the distribution of the metrics in the studied systems. When possible, we analyze varying degrees of aggregation (*e.g.,* methods, classes and hierarchies) in order to evaluate how the results hold at each level of granularity, and to avoid ecological fallacies, which can occur when one studies the data at the wrong abstraction level [PFD11].

3.5 Experimental Results

In this section we discuss in turn the research questions that we proposed in Section 3.1. For both Smalltalk and Java 99% of the inspected projects define polymorphic methods and polymorphic message sends. As for duck typing, 99% of the inspected Smalltalk projects define duck-typed methods and duck-typed message sends.

3.5.1 Implementing polymorphism

Figure 3.4 shows the proportion of polymorphic methods, *i.e.,* methods whose selectors are polymorphic for both Smalltalk (left) and Java (right). This information is then aggregated to the level of classes and hierarchies.

Figure 3.4 shows that, for Smalltalk:

- **At least one out of four methods (31%)** in the project implements a polymorphic selector.
- **63% of all classes** in the project implement at least one of those methods
- **Almost all class hierarchies (97%)** in the project include at least one polymorphic selector

Figure 3.4 shows that for Java the numbers are lower but they still reveal that:

- **At least one out of five methods (24%)** in a project implements a polymorphic selector
- **Almost half (44%) of the classes** in the project implement at least one of those methods
- **More than three quarters (76%) of all class hierarchies** in the project include at least one polymorphic selector

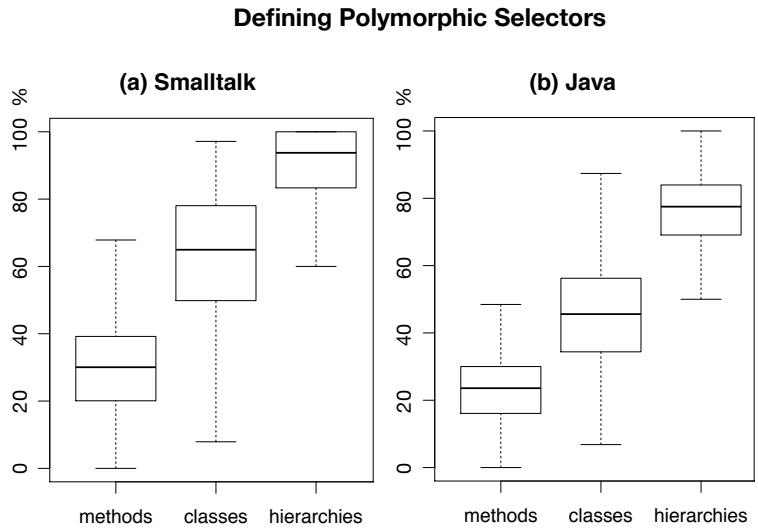

Figure 3.4: Distribution of the proportions of methods, classes and hierarchies defining polymorphic selectors: in (a) Smalltalk and (b) Java

Based on this data, we can now answer the first research question in the chapter by stating that:

> In a majority of projects in both Smalltalk and Java more than a quarter of the methods are implementations of polymorphic selectors. Polymorphic methods are more present in Smalltalk than in Java.

3.5.2 Using polymorphism

In Figure 3.5 we present the proportion of all message sends that are polymorphic as well as the proportion of all methods and classes defining at least one polymorphic message send (on the left for Smalltalk, on the right for Java).

For Smalltalk, Figure 3.5 shows that:

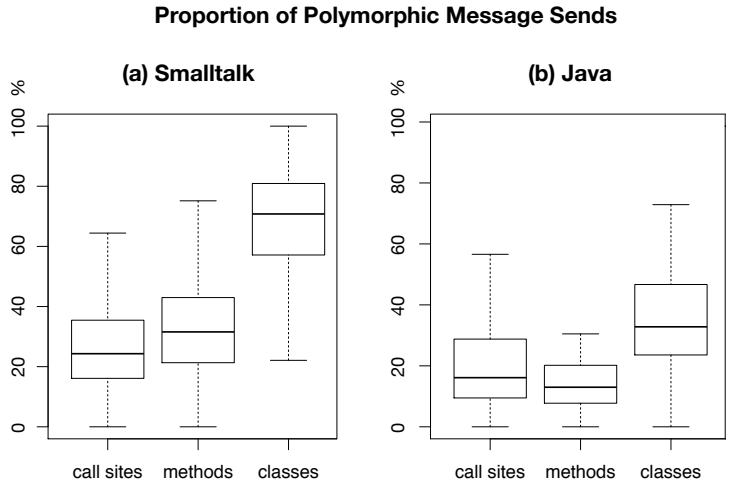

Figure 3.5: Distribution of the proportion of polymorphic message sends, as well as methods and classes having them: in (a) Smalltalk and (b) Java

- **A quarter of all the message sends** in the project (24%) are considered to be polymorphic
- **A third of the methods** in a project (32%) contain a message send considered to be a polymorphic
- **Three quarters of the classes** in a project (78%) contain a polymorphic message send.

For Java, Figure 3.5 shows that:

- **16% of all message sends** in a project cannot be resolved at compile time using the analysis we have explained in the Terminology section
- **12% of all methods** in a project include a message send which is considered not capable of being resolved at compile time
- **30% of classes** in a project have at least one polymorphic message send

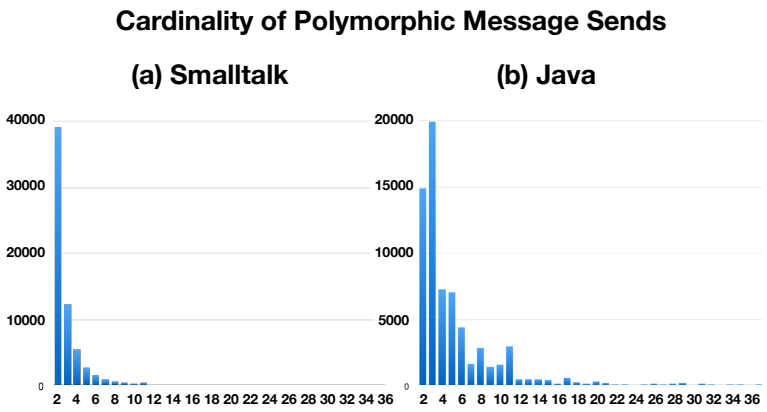

Figure 3.6: The cardinality of the polymorphic message sends in the Smalltalk and the Java corpus, respectively

Surprisingly, in Java the proportion of methods defining a polymorphic message send is lower than the proportion of polymorphic message sends. We hypothesise that polymorphic message sends cluster in methods, although this would have to be verified in future studies.

We can now answer the second research question in the chapter as follows:

> For a majority of the projects, more than one in seven (for Java) and one in five (for Smalltalk) of the message send in a system are considered to be polymorphic.

This means that the issue presented in the example in Section 3.1 is frequently encountered by developers working in both languages. Tools to support program understanding in the presence of polymorphism are even more important in the Smalltalk context than in the Java context.

3.5.3 Cardinality of polymorphic message sends

The cardinality of a polymorphic message send is defined as the size of the set defined earlier as $impl(t, msg(ms))$ (3.13), *i.e.,* the number of methods

implementing that message.

The distribution of the cardinality of polymorphic message sends in Smalltalk is presented in Figure 3.6, on the left. We observe that:

- More than 75% of polymorphic message sends have cardinality two or three.
- Most of the message sends (90%) that cannot be resolved at compile time using the implemented analysis have strictly less than 6 candidates.

Figure 3.6 shows on the right the results of running a similar analysis for Java. Based on the analysis of 71K polymorphic message sends in the corpus, we observe that:

- There are fewer message sends with two candidates than with three candidates.
- More than 50% of the message sends have a cardinality of two or three
- More than 75% of polymorphic message sends have cardinality less than seven
- Most of the message sends (90%) have less than 12 candidates

In some cases, there are message sends with a very large cardinality. Table 3.1 presents several of the extreme cases we have investigated. In all the cases we have more than 100 potential selector implementations for a message send. We can observe some design patterns that are responsible for this, including Visitor (for jruby) and Command (for ant).

Table 3.1: The largest cardinalities in the Java corpus

System	Method Name	Cardinality
weka-3.7.5	RevisionHandler.getRevision()	545
spring-3.0.5	InitializingBean.afterPropertiesSet()	216
ant-1.8.2	Task.execute()	201
weka-3.7.5	CapabilitiesHandler.getCapabilities()	167
jruby-1.5.2	Node.interpret(Ruby,ThreadContext,...)	148

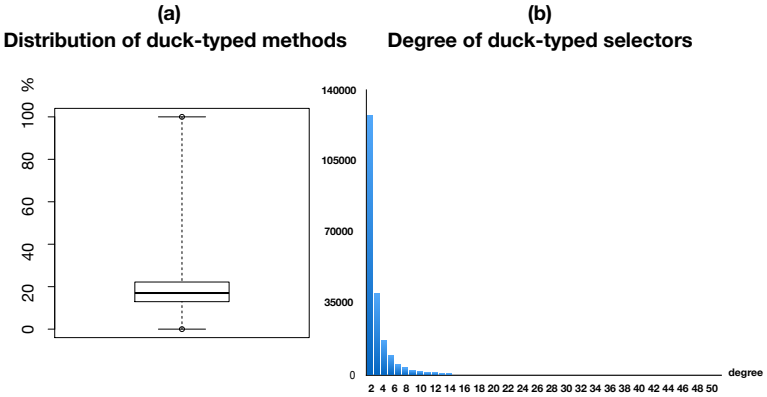

Figure 3.7: Implementing duck-typed selectors in Smalltalk

Several of the polymorphic message sends with large cardinalities (e.g. the ones in Table 3.1) also happen to occur a significant number of times. This is probably the result of a given message send occurring in multiple places in a given project.

To finally answer the research question, we conclude:

> In both languages a strong majority (75%) of the polymorphic message sends have a cardinality of up to six and a vast majority (90%) have a cardinality of less than twelve. Corner cases can have hundreds of candidates.

3.5.4 Implementing duck typing

Figure 3.7 (a) shows the proportion of methods in Smalltalk whose selectors are duck-typed. We can observe that:

- for half of the analysed projects **at least one of six methods (17%)** in a project implements a duck-typed selector
- most of the analysed projects contain **up to 22%** of duck-typed methods

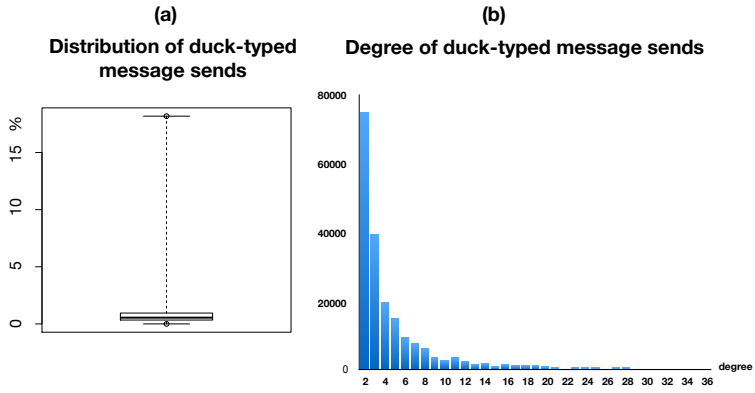

Figure 3.8: Using duck-typed selectors in Smalltalk

However, there are strong outliers with more than 50% of methods implementing duck-typed selectors.

We have also measured the degree of each of the duck-typed selectors, *i.e.,* $|intr(s)|$. The results are presented in Figure 3.7 (b).

We can observe that:

- 75% of duck-typed signatures are implemented in up to five distinct hierarchies, *i.e.,* have a degree of up to five
- some outliers have more than 200 cross-hierarchy implementations

Based on this data, we can now answer the fourth research question in the chapter by stating that:

> In half of the projects in Smalltalk more than one sixth of the methods represent implementations of duck-typed selectors. Three quarters of duck-typed selectors have cardinality of up to five.

3.5.5 Using duck typing

In Figure 3.8 (a) we present the proportion of the message sends in Smalltalk that are duck-typed.

We can state that:

- In half of the analysed projects, more than 0.55% of message sends are duck-typed
- In around one fifth of the analysed projects, more than 1% of message sends are duck-typed.
- Almost all of the analysed projects contain up to 15% of duck-typed message sends.
- There are strong outliers where around 18% of message sends are duck-typed.

We have also measured the degree of duck-typed message send, *i.e.,* the number $|intr(msg(ms))|$. Our findings are presented in Figure 3.8 (b). We observe that:

- 37% of duck-typed call sites have degree of two
- about 30% of duck-typed call sites have degree of three or four
- about 28% of call sites have degree varying from five to 17

There are strong outliers with degree of more than 200.
We can now answer the fifth research question in the chapter with:

> All projects contain to some extent duck-typed message sends. Most of them have up to 1% of message sends which can be duck-typed. Degrees of these message send vary from two to more than 200.

3.6 Threats to Validity

Construct Validity. The best way to detect polymorphic and duck-typed methods would be to use a combination of both static and dynamic analysis. This threatens the validity of our study since we are unable to know whether the resulting methods are parts of "hot spots" in the source code, or whether they are actually never executed. However, our goal was to look at the

problem from the perspective of the IDE where only static information is normally available.

The static analysis algorithms that we have used represent other threats to the validity of our results. In this analysis we used the CHA algorithm (*Class Hierarchy Analysis*) [DGC95] for Java and a modified version for Smalltalk to calculate the candidates for the message sends regarding polymorphism. We have encountered cases where there are more than 100 method candidates for a polymorphic message send. With more advanced techniques of static analysis, like the RTA algorithm (*Rapid Type Analysis*) [BS96], it might be possible to get more precise results. While CHA finds the possible method candidates at the message send based only on the declared type of the receiver, RTA takes into account the set of all classes instantiated until the corresponding moment in the analysis. It could be that more advanced analyses would provide more precise results. However, our goal was to assess the criticality of polymorphism on program understanding from the perspective of a developer using common IDEs, and the burden it poses on simple static analysis techniques.

Our decision to treat a message send in Java in the same way, regardless of whether the statically-declared type of the receiver is an interface or a class may influence our research, since we also include abstract methods in the results. However, we wanted to estimate the challenges that are faced by a developer when analysing the code. Abstract methods are usually included in the results of static analysis, *e.g.,* navigation action, since present IDEs do not typically include advanced analyses.

One final threat to construct validity concerns possible imprecisions in detecting polymorphic methods, due to cross-hierarchy polymorphism (*i.e.,* duck typing) in Smalltalk, since we are not able to statically determine the type of the receiver. Due to cross-hierarchy polymorphism, the results may contain false positives, *i.e.,* message sends marked as polymorphic, whereas they are not.

Internal Validity. We consider only user-defined polymorphism, and not polymorphism defined in libraries being employed by the analysed projects, nor the usage of these polymorphic library methods. Moreover, we consider polymorphism only within the boundaries of a system, without taking into consideration library classes being extended in the subject systems, thus there might be more user-defined polymorphic methods whose superclass implementations are outside of project boundaries.

External Validity. Since our study features only open-source projects, we

cannot generalise our findings to industrial projects. For the Smalltalk corpus, we only considered projects that are found in the SqueakSource repository. Although SqueakSource was at that time the *de facto* standard source code repository for Squeak and Pharo developers, we cannot be sure to what extent the results generalise to Smalltalk code outside of SqueakSource, such as Smalltalk code produced by VisualWorks developers. We only take into account Smalltalk projects with more than 50 classes to filter out projects that might be toy or experimental projects. We believe such filtering increases the representativeness of our results, however, it might also impose a threat. Similarly, our corpus of Java systems contains only open-source code, and is built based on the availability of systems. As such, we cannot make strong claims about its representativeness. It does however contain popular applications, such as ArgoUML, components of Apache, FindBugs, etc.

The two sub-corpora exhibit different characteristics: notably, polymorphism is much more prevalent in Smalltalk — a "pure object-oriented" language — than in Java. Extending the study to other OO languages may yield other insights.

3.7 Discussion

The study presented above only considers the prevalence of polymorphism in open source software. As such, it represents only the first step. In this section we consider a series of further research questions that the results of the study raise.

How To Improve Simple Analysis Techniques? While there is not a big difference between the portion of methods that implement polymorphic selectors in Smalltalk and Java, there is quite a difference between message sends that our algorithm found to be polymorphic in both corpora. Since in Smalltalk there is no notion of static type of the receiver, some of the found polymorphic message sends may not actually be polymorphic, due to the receiver always being of the same type. Again, to be able to infer the type for the receiver, static analysis encounters an obstacle in the form of polymorphism. There is a research question of how static analysis techniques may be improved in the presence of (cross-hierarchy) polymorphism.

Duck Typing. This kind of language idiom is usually encountered in dynamically-typed programming languages, but can be simulated also in

statically-typed languages with the use of interfaces. Instances of classes that model distinct domain objects, but implement the same interface, might be interchangeably used whenever the interface type is expected. Hence, an interface selector may be implemented in both classes, even though the intended behaviours may differentiate. These implementations may be considered as duck-typed. It would be important to empirically quantify the impact of duck typing on program understanding. We expect duck typing to actually have a more drastic impact on program comprehension. While we have measured separately polymorphism and duck typing usage, it is more likely that these two dimensions of polymorphism are combined in the code implementation. That is, we expect to find duck-typed selectors being also overridden, hence heavily influencing type inference analysis. As such, they present an even greater threat to program comprehension and static analysis techniques.

Static and dynamic detection: What is the Ground Truth? As we have already explained, we have used variations of CHA algorithm for both Java and Smalltalk, in order to calculate the polymorphism metrics. CHA represents one of the simplest static algorithms for call graph construction, and it is commonly plugged in the IDE analysis. However, even with the most advanced static analysis, there will be cases where a message send cannot be resolved statically. The downside of static analyses are the false positives, *i.e.,* potential polymorphism might not occur in practice [RMR03, RKG04, Dmi04], while dynamic analyses suffer from false negatives, *i.e.,* some polymorphic calls may be missed in any given run [PTP07]. This leads us to the question of the best technique to detect polymorphic message sends: what is the ground truth, and which technique yields the best approximation?

Impact on Program Understanding. Given that the presence of polymorphic message sends is so high, based on the results of the second research questions, it would be important to discover to what extent polymorphic message sends pose a problem for program understanding. These results also bring up the question of how polymorphism influences the quality of the source code, *e.g.,* whether it is more comprehensible for developers to use type checking or polymorphism.

On the other hand, it would be important to assess the differences in program comprehension in statically- and dynamically-typed software which exercise polymorphism to the same degree. This would provide a precise estimation of how much static type information helps developers

to understand polymorphic code.

With the cardinality of a polymorphic message send measure we tried to estimate the complexity of understanding a given polymorphic message send. However, the cardinality is a preliminary measure. For example, often the methods that could be invoked by a given message send will also contain polymorphic message sends in their turn. A user would have to follow such a *polymorphic call chain* to fully understand the software.

A better proxy for the difficulty of understanding a polymorphic message send would take into account also a tree-like structure of polymorphic call chains. Such a metric would be the equivalent of cyclomatic complexity computed on the call graph induced by the compound polymorphic calls. In preliminary studies we observed some extreme polymorphic call chains that span dozens of methods.

Polymorphic call chains could be combined with other measures to eventually quantify the impact of polymorphism on comprehension. Eventually, empirical studies would have to validate such metrics.

3.8 Conclusion

We have performed an empirical study of polymorphism on two corpora of open source systems written in Smalltalk and Java, respectively. We found that polymorphism is frequently used in both languages: nearly all projects we analysed take advantage of polymorphism by implementing polymorphic selectors and by invoking such selectors.

We learned that in Java, half of the polymorphic message sends have a cardinality of two or three, and three quarters have a cardinality of less than seven. Smalltalk uses polymorphism to a greater extent: more than 60% of all classes implement a polymorphic selector in Smalltalk projects, while around 40% do the same in Java projects. As for methods, 31% of methods in Smalltalk implement polymorphic selector compared to 24% in Java. Since one fifth of all message sends in Smalltalk projects are polymorphic, and one third of all methods are implementations of a polymorphic selector, solving the problems associated with understanding polymorphic message sends is of higher priority for Smalltalk than for Java.

Even though there is a significant difference in polymorphism usage in the two corpora it is not as drastic as the consequent difference in program comprehension difficulties in dynamically- and statically-typed languages [SMDV08, KBR14]. We may suppose how big is the impact of

the lack of static type information on program comprehension.

As another dimension of polymorphism, we have measured the presence of cross-hierarchy polymorphism in Smalltalk. Our findings are that all of the analysed projects exercise to some extent duck typing, and that most of the duck-typed message sends have cardinality of up to 17. On the other hand, methods that implement a duck-typed selector count for about one fifth of all the methods in a project.

These findings also indicate clearly the extent of difficulties polymorphism poses on static analysis techniques, and allow us to estimate the source of difficulties of simple type inference algorithms.

4

Static class usage frequency heuristics

4.1 Introduction

There are various techniques that attempt to reconstruct the types of variable in code. While some of them rely solely on the information collected statically from the code, *i.e.,* without program execution, others collect information at run time, and feed it back to the algorithm. Some approaches tend to be more precise, while others trade precision for speed. Some approaches depend on typing the whole program, while others handle an isolated variable on its own. Many dimensions may be drawn in order to classify the field of type inference algorithms.

To offer precise type information, a type inference algorithm should analyse data flow, which is heavily intertwined with the control flow of the software. One cannot reason about the one without the other. While the execution of the program offers precise control- and data-flow information, it offers just a narrow view of the possible execution paths [PTP07], and introduces execution overhead.

Static type inference analysis is less expensive, but suffers from the

problem of false positives. In general, precise data-flow analysis is NP-hard for either flow-insensitive [Sus97] static analyses, or flow-sensitive [LR91, Lan92, Ram94] ones. Any attempt to construct the type of a variable by static analysis suffers from some level of imprecision. As expected, more precise algorithms depend on the analysis of the whole program, making it slow and expensive.

Simple approaches tend to pose certain constraints on a variable (or a program expression) whose resolution would result in the set of possible types for that variable (expression). These constraints constitute, for example, of the set of messages sent to the variable, *i.e.,* locally used interface of the variable, or the possible type flow at run time. The main purpose of these approaches is to be used for program comprehension and provide a developer with fast information. Due to their simplicity, and sometimes slightly naive approach, they are not intended to be used for the compiler optimisation. Because of the plain analysis their precision is hampered by the usage of object-oriented features, like different kinds of polymorphism and dynamic features. For example, by statically tracking down the set of messages sent to the variable, it is possible to uniquely determine possible types for less than 60% of variables [PMW09]. The usage of popular interfaces for the rest of the variables generates a large number of false positives, *i.e.,* classes inferred as potential types for a variable, but not representing its actual type at run time. This is a direct consequence of high usage of polymorphism combined with duck typing in object-oriented code. However, these approaches are mainly intended for program comprehension, thus offer a reasonable compromise between speed and precision.

One of the problems for simple algorithms is that popular interfaces result in large number of possible types for a variable, therefore introducing ambiguity. Presenting the list of possible types for a variable in no particular order, where most of them cannot actually represent the variable type during run time, is not helpful to a developer. Meddling with the algorithm would increase its complexity, and subsequently the time needed to perform the analysis. That is why we propose ordering the resulting set of classes to indicate those more likely to be correct. All of these classes are more or less likely to represent the actual type of the variable at run time. Regardless how complicated is the control flow of a software, all the instances to which the variable may be bound at run time must be created somewhere in the code. The usual way is to invoke a constructor of the desired class, leading us to suppose that the more frequently the class is instantiated throughout

the code, the more likely it is that it will represent a variable type at run time.

Classes are not only used to instantiate a new object, but also to invoke class methods that perform utility functions, which may also result in instance creation. We hypothesise that class usage frequency serves as a reliable proxy for the likelihood that a variable may be bound to an instance of that class at run time. Thus, we explore two heuristics for ordering the list of possible types for a variable: according to how frequently classes are instantiated in the code and according to how frequently class name occurs in the source code. The information about class presence in the source code is easy to retrieve, thus preserving the proposed approach as simple and swift. Types of a variable are primarily inferred statically based on the messages sent to it and from the assignments to the variable. We have used *RoelTyper* [PMW09] for this purpose. The approach is explained in details throughout the section Section 4.3.

We have implemented a proof-of-concept prototype in Pharo Smalltalk and used it to evaluate the proposed heuristics. First, the evaluation showed an overhead of 4.56%, due to the sorting function, which we deem acceptable. The most of the overhead comes from the calculations of the value for each class, which we use to sort them. Second, it proved that the implemented heuristics led to a significant improvement when compared to the basic approach. Third, the heuristic that showed better results, *i.e.,* the heuristic based on the frequency of class occurrence in the code, was compared to an existing improvement of the same underlying type inference technique, and introduced an improvement of the results with fewer requirements. The improvement is achieved in 58.6% of the cases.

The rest of the chapter is organised as follows: We start by giving an overview of the problem in Section 4.2. Next we define the used terminology and implemented heuristics in Section 4.3; Section 4.4 shows results of the evaluation of the prototype. We then describe potential threats to the validity in Section 4.5 before concluding in Section 4.6.

4.2 Overview

To better understand the contribution of the chapter, let us expend the example presented in Chapter 1 (Listing 1). The example is repeated in Listing 3.

```
1 GLMLoggedObject subclass: #GLMPane
2    instanceVariableNames: '... presentations ...'
3    classVariableNames: ''
4    category: 'Glamour-Core'
5
6 GLMPane>>update
7    ...
8    self presentations do: [ :each | each update ]
```

Listing 3: The run-time type of the argument `each` cannot be statically detected by the traditional approach

Lines 1-4 define a new class `GLMPane` which has an instance variable `presentations`, while lines 6-8 define a method named `update` that performs an update operation on each of the elements of the instance variable `presentations`.

Let us suppose that the developer wants to know the possible type of the argument `each` in Listing 3, as she is interested to know the types of the individual elements added to the instance variable `presentations`. Since Smalltalk is a dynamically typed language, the developer cannot determine statically the potential type of the argument `each`.

The simple (standard) approach to infer variable types for this variable would be to traverse the list of messages sent to it and find all the classes in the image[1] that understand the interface of the variable, *i.e.,* in this case `update` selector.

Using the traditional approach, the developer will be offered the list of 121 possible classes[2] grouped by their twenty hierarchies. These classes are represented in no particular order. Thus, she will obtain a list of 121 possibilities, with no particular knowledge of how likely it is that any of the classes is the correct one. Evidently, this information is not helpful to the developer. Even if the developer would restrict the search of the types to the package in which the class `GLMPane` occurs, it would still leave her with 52 classes to examine, organised in three hierarchies.

In order to narrow down the list, we note that `each` is an element of `presentations`, which is presumably a collection. By further inspection of the class `GLMPane`, we can see which other messages are sent to these elements.

[1]The term “Pharo image” is used to denote a snapshot of the running Pharo system.

[2]The system used with this example is Moose 5.0, a platform for software and data analysis based on Pharo. The actual number of implementations may vary in the other systems.

```
GLMPane>>resetAnnouncer                                          9
   super resetAnnouncer.                                         10
   self presentations do:                                        11
      [ :each | each resetAnnouncer]                             12
                                                                 13
GLMPane>>addPresentationSilently: each                           14
   ^ presentations                                               15
      add: (each pane: self; yourself)                           16
```

Listing 4: Access to the elements of the variable `presentations`

In Listing 4 we observe in line 12 that message `resetAnnouncer` is sent to the elements of `presentations`. In line 16 we observe that the argument `each` of the method named `addPresentationSilently:` is added to the instance variable `presentation`. This argument needs to understand messages `pane:` and `yourself`, hence elements of the variable `presentations` need to understand them as well. After taking this information into account, there are still 63 possible types for the elements of the variable `each`. (Bear in mind that this kind of analysis requires control and flow analysis, thus is more difficult to perform automatically.)

```
GLMPane>>addPresentations: aCollection                           17
   self notingPresentationChangeDo: [                            18
      aCollection do: [ :each |                                  19
         self addPresentationSilently: each                      20
      ]                                                          21
   ]                                                             22
                                                                 23
GLMPresentation>>pane                                            24
   ^ pane ifNil: [                                               25
      pane := (GLMPane named: 'root'                             26
                        in: GLMNoBrowser new)                    27
               addPresentationSilently: self;                    28
               yourself                                          29
   ]                                                             30
```

Listing 5: Senders of the method `addPresentations:`

Analysing senders of `addPresentationSilently:` does not yield much insight into the type of the argument `each`. In lines 20 and 28 in Listing 5 we can see that the method is invoked twice within the source code. Line 20 provides no information about the elements `each` of the

method argument `aCollection`. In line 28, the argument that is passed is `self`, which is an object either of type `GLMPresentation` or any of its subtypes, which leaves us again with 63 possible types for the method argument in line 6 in Listing 3. As previously explained, this kind of analysis depends on control flow, hence is difficult to perform automatically in dynamically typed languages [Sus97].

One can reasonably suppose that the name of the instance variable `presentations` reveals the type of its elements, since the class `GLMPresentation` is present in the project. In this case the name is not of a much help, since the class `GLMPresentation` has 63 subclasses.

We propose to exploit information about the usage of the classes representing possible types of the variable `each` to highlight which classes are more likely to represent the variable's actual type(s) at run time. We argue that the degree of usage of the classes throughout the source code is strongly related to the likelihood of the class being used as a type for a particular variable. We have developed a couple of heuristics based on the different ways of class usage with the aim to improve type inference techniques.

One of the heuristics sorts the possible types based on the number of occurrences of the class name throughout the image. In the example from the Listing 3, the list of possible types for the argument `each` contains 121 classes in the following order, sorted by the frequency of the class names occurring in the image:

1. `GLMTabulator`

2. `GLMCompositePresentation`

3. `GLMFinder`

4. `GLMPaneAdded`

5. `MooseFinder`

...

121. `PaneAbstractLine`

As a result, a developer will be issued with the above presented list of 121 possible types in the specified order. The actual type of the variable at run time is `GLMTabulator` in multiple runs of the system, which is at the top of the list. We argue that it is possible to introduce accurate information

about the type of a variable with such heuristics, avoiding false positives, and that this will provide more insightful information to developers for program comprehension.

4.3 Heuristics and Approaches

4.3.1 Terminology

To explain the heuristics, we introduce a simple set-theoretic model in Figure 4.1 that captures key properties for the entities shown in the UML diagram in the Figure 4.2.

$$\mathit{msg} : V \to \mathcal{P}(S) \tag{4.1}$$
$$\mathit{sel} : M \to S \tag{4.2}$$
$$\mathit{def}_m : M \to C \tag{4.3}$$
$$\mathit{sup} : C \to C \cup \{\mathit{null}\} \tag{4.4}$$
$$\mathit{assign\_types} : V \to \mathcal{P}(C) \tag{4.5}$$
$$\mathit{undr} : C \cup \{\mathit{null}\} \times S \to \{true, false\} \tag{4.6}$$

Figure 4.1: The core model.

Given a target program language, C is the domain of all classes, M is the domain of all methods, S is the domain of all selectors. V is the domain of all variables, including instance variables, method arguments and local variables.

Each variable v has a (possibly empty) set of messages $\mathit{msg}(v)$ sent to it in its lexical scope (4.1) either directly or through the getter method. Note that we consider the lexical scope for instance variables only to be the methods of the class in which it is defined, but not its subclasses. In case of the instance variables inherited from superclass, we choose to treat them as instance variables defined in the subclass. The implications of this decision are discussed in Section 4.5. We call this set of messages the *interface* of the variable v. Each method m has a unique selector $s = \mathit{sel}(m)$ (4.2), and is defined in a unique class $c = \mathit{def}_m(m)$ (4.3). Each class c has a unique

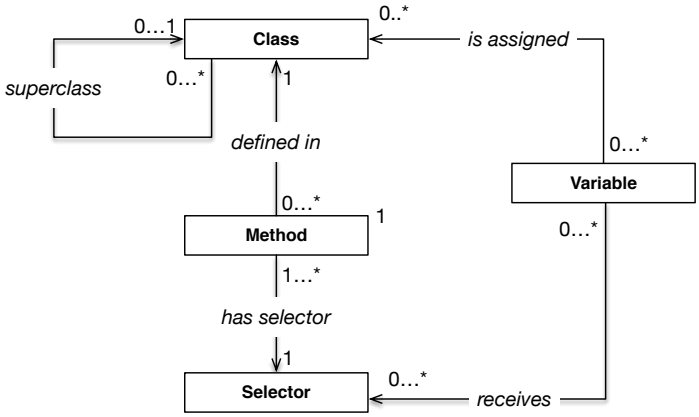

Figure 4.2: The core model in UML

superclass $c' = sup(c)$ (4.4). We define the superclass of `Object` to be *null*, *i.e., sup*(`Object`) = *null*.

Consider the example class hierarchy in Figure 4.3.

In this example there is a class `RTGlobalBuilder` with an instance variable named `properties` and methods `addProperty:`, `execute` and `initialize`. Within these three methods messages sent to the instance variable `properties` are

$$msg(\texttt{properties}) = \{\texttt{add:}, \texttt{do:}\}$$

Also, each variable v may have one or more assigned types $c \in assign\_types(v)$ (4.5) if the variable v is the left side of an assignment where the right side of the same assignment is a message send to a class that results in creating a new object, *i.e.,* is a call to a constructor or this newly created object has been assigned to the variable via setter method. Returning to the example, in the method `RTGlobalBuilder>>initialize` there is an assignment to the instance variable `properties` of the newly created object of type `OrderedCollection`, which means that

$$assign\_types(\texttt{properties}) = \{\texttt{OrderedCollection}\}$$

We have used a couple of heuristics to guess the type of the expression result assigned to the variable, as done with RoelTyper [PMW09]. These

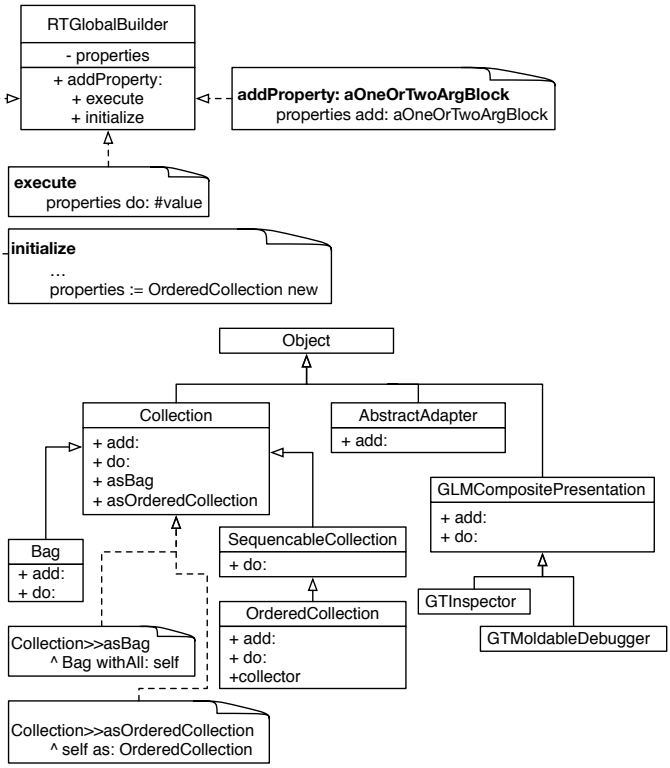

Figure 4.3: Sample class hierarchy

heuristics are listed in the Table 4.1. Having multiple assignments to the same variable is possible, but it is beyond the scope of our small example.

We can now query the model to ascertain the set of possible types for every variable. Each class c can either understand the selector s or not (4.6). The class c understands selector s if this class defines a method $m \in \mathit{def}_m^{-1}(c)$ such that $\mathit{sel}(m) = s$ or its superclass $\mathit{sup}(c)$ understands it (4.7). We also define that $\mathit{undr}(null, \text{s}) = false$. The interface of the class c is a set $\mathit{intr}(c)$ of all the selectors s that class c understands (4.8). The class c is a possible type for the variable v if class c understands the interface of the variable v (4.9). For a set of classes C' we infer as roots the classes

$$undr(c, s) = s \in sel(def_m^{-1}(c)) \vee undr(sup(c), s) \tag{4.7}$$

$$intr(c) = \{s \in S | undr(c, s)\} \tag{4.8}$$

$$sel\_all\_types(v) = \{c \in C | msg(v) \subseteq intr(c)\} \tag{4.9}$$

$$roots(C') = \{c \in C' | \forall n > 0, sup^n(c) \notin C'\},\ C' \in \mathcal{P}(C) \tag{4.10}$$

$$sel\_types(v) = roots(sel\_all\_types(v)) \tag{4.11}$$

Figure 4.4: Computing possible types for a variable.

from the same set without having a superclass in C' (4.10). At the end, we infer roots of hierarchies to which possible types for a variable may belong (4.11). To this set of classes, as well as to the set *assign_types*(v) we will apply ordering. For now we focus on the set of root classes, since RoelTyper [PMW09], as well as the technique built on top of it [SLN14] both infer only possible hierarchies. We will later discuss the set of all classes that may represent a variable type.

In the example in Figure 4.3 we see that the class `Collection` understands the selector `add:`, as do all of its subclasses and the class `AbstractAdapter`, while *undr*(`RTGlobalBuilder`, `add:`) = $false$.

We can now calculate the interfaces of the classes. For the sake of brevity, we only list those messages that are relevant for this example, rather than the complete interfaces that hold more than 400 selectors:

intr(`Collection`) = *intr*(`Bag`) = *intr*(`SequenceableCollection`) = {`add:`, `do:`, `asBag`, `asOrderedCollection`}

intr(`OrderedCollection`) = {`add:`, `do:`, `collector`, `asBag`, `asOrderedCollection`}

intr(`AbstractAdapter`) = {`add:`}

intr(`GLMCompositePresentation`) = *intr*(`GTInspector`) = *intr*(`GTMoldableDebugger`) = {`add:`, `do:`}

From the previous equations, we see that

msg(`properties`) $\subseteq$ *intr*(`Collection`)

Expression	Inferred type
$x = y \quad x == y \quad x \sim= y \quad x < y \quad x > y \quad x <= y \quad x >= y \quad x \ = y$	Boolean
$x\ msg\ y$, where msg is any of the arithmetic, logarithmic or trigonometric functions or functions used to round a number	Number

Table 4.1: Heuristics used to infer the type of the expression

msg(`properties`) $\subseteq$ *intr*(`Bag`)
msg(`properties`) $\subseteq$ *intr*(`SequenceableCollection`)
msg(`properties`) $\subseteq$ *intr*(`OrderedCollection`)
msg(`properties`) $\not\subseteq$ *intr*(`AbstractAdapter`)
msg(`properties`) $\subseteq$ *intr*(`GLMCompositePresentation`)
msg(`properties`) $\subseteq$ *intr*(`GTInspector`)
msg(`properties`) $\subseteq$ *intr*(`GTMoldableDebugger`)

Classes that understand the interface of variable `properties` all belong to two distinct hierarchies with the root classes `Collection` and `GLMCompositePresentation`, thus

sel_types(`properties`) =
`{Collection, GLMCompositePresentation}`

4.3.2 Heuristics

The first intuition was that the classes instantiated the most throughout the source code are more "present" in the source code, and they are more likely to represent the types of the variable than classes that are less "present" in the source code. Classes can be used to instantiate new objects or they can be used "on their own", as independent objects. Hence, we have implemented and evaluated two possible heuristics:

1. Class instantiation heuristic

2. Name occurrence heuristic

We continue by explaining the evaluated heuristics used for ordering the possible types of a variable v. We use each heuristic to order separately two sets of classes: *assign_types*(v) and *sel_types*(v).

Class instantiation

The intuition is that the more frequently the class is instantiated throughout the source code, the more likely it is that it represents the type of a variable. This heuristic is founded on the calculation of the occurrences of class instantiation throughout the source code.

The usual way to create a new object in Smalltalk is to send a message `new` to a class. Besides the methods with the selector `new`, all the methods that belong to any of the protocols[3] `initialize`, `initialization` and `instance creation` are considered to be constructors, *i.e.,* result in the creation of a new object. So, we count all the occurrences of class usage like `OrderedCollection new` or any other message send to a class that results in invoking a method from any of the previously mentioned protocols.

Returning to the example in Figure 4.3, the only place in the source code where a class is instantiated is within the method `RTGlobalBuilder>>#initialize`, where a new object of the type `OrderedCollection` is created and assigned to the instance variable `properties`. Based on this information, we can count the corresponding frequency of class instantiation for each class:

class_inst(`Collection`) = *class_inst*(`Bag`) =
class_inst(`SequenceableCollection`) = 0

class_inst(`OrderedCollection`) = 1

class_inst(`GLMCompositePresentation`) =
class_inst(`GTInspector`) = *class_inst*(`GTMoldableDebugger`) = 0

As we infer root classes based on the variable interface, we calculate the value based on which *sel_types*(v) are sorted as following: for class $c \in$ *sel_types*(v) the value of the class is the sum of *class_inst*(c') for each subclass c' of the class c (4.12). As for the classes that belong to the set *assign_types*(v), we consider them as *truthful*, *i.e.,* without taking

[3]Methods in Smalltalk are organised in protocols, *i.e.,* groups of related methods.

subclasses into account. Thus, these classes are sorted purely based on their own values, *i.e., class_value_assign*(c), for class $c \in$ *assign_types*(v) is equal to *class_inst*(c) (4.13).

$$\textit{class\_value\_sel}(c) = \sum_{\substack{c=\textit{sup}^n(c'),n\in N \\ c'\in C}} \textit{class\_inst}(c') \tag{4.12}$$

$$\textit{class\_value\_assign}(c) = \textit{class\_inst}(c) \tag{4.13}$$

Figure 4.5: Calculating class values for selector and assignment types

We can now calculate values for classes in *sel_types*(`properties`):

class_value_sel(`Collection`) = 1
class_value_sel(`GLMCompositePresentation`) = 0

We can now sort the possible types for the variable `properties`. As we have mentioned at the beginning of Subsection 4.3.2 for each variable v we sort independently the lists *assign_types*(v) and *sel_types*(v). In the example, the list *assign_types*(`properties`) has only one element, so there is no need for sorting. The list *sel_types*(`properties`) has two elements that will be sorted as follows:

1. `Collection`
2. `GLMCompositePresentation`

Name occurrence

Since a class can be used as an object itself, *e.g.,* to invoke a class method that does not necessarily need to be a constructor, in this heuristic we focus on all the places in source code where a class name is used, except for class definitions. We count these occurrences and use the relative frequency to sort the possible types of a variable.

In our example in Figure 4.3, we encounter two occurrences of the class `OrderedCollection`: in the methods `RTGlobalBuilder>>#initialize` and `Collection>>#asOrderedCollection`. Hence,

name_occ(`OrderedCollection`) = 2

The class `Bag` is used as a receiver for a message send `withAll:` in the method `Collection>>asBag`, hence

$$name\_occ(\texttt{Bag}) = 1$$

Classes `Collection` and `SequenceableCollection` are mentioned nowhere in the code, as the classes `GLMCompositePresentation`, `GTInspector`, `GTMoldableDebugger`, except for its declaration, which we do not count, so

$$name\_occ(\texttt{Collection}) = name\_occ(\texttt{SequenceableCollection}) = name\_occ(\texttt{GLMCompositePresentation}) = name\_occ(\texttt{GTInspector}) = name\_occ(\texttt{GTMoldableDebugger}) = 0$$

Thus, values of the classes inferred as types based on the message sends to the variable `properties` are

$$class\_value\_sel(\texttt{Collection}) = 3$$
$$class\_value\_sel(\texttt{GLMCompositePresentation}) = 0$$

As in the previous heuristic, there is only one assigned type to the variable `properties`, so there is no need for sorting the list *assign_types*(`properties`). However, the sorted list *sel_types*(`properties`) will look like the following:

1. `Collection`
2. `GLMCompositePresentation`

4.3.3 Assigned types vs. selector types

In our evaluation we give the priority to the assigned type, since we believe that this information is inserted by a developer with a high accuracy. As a result of the inference process, for each variable v two lists will be presented, namely, *assign_types*(v) and *sel_types*(v), so that a developer has a notion of types explicitly being assigned to the variable, and the types that are implicitly determined. If there are no assignment types for a variable v, only selector types are presented and vice versa.

4.3.4 Approaches

In our opinion, it is important for the developer to know the hierarchy of classes to which the run-time type of the variable may belong, but also to have a notion of the specific type of a variable. For that reason, we have developed both a hierarchy-based approach and a class-based one. To the best of our knowledge, this is one of the most simple algorithms that tries to infer the precise type for variables, and not just the class hierarchy. As the tool is intended mainly to assist developers in program comprehension, we consider it important to infer both possible classes and hierarchies to which the type of the variable may belong. Names of a root class and a subclass may be very different and developer is sometimes expecting a class of a certain name [KBR14]. For that reason, we deem it important to present her with the list of root classes as well as the list of all classes.

For each variable v both approaches collect the messages sent to the variable v. The main difference thereafter is in computing the set of classes that understand the interface of the variable, that it *sel_types*(v).

Hierarchy-Based Approach

This approach is explained in the example in Figure 4.3 throughout Subsection 4.3.1. Let us remember that in the example in Figure 4.3 the interface of the instance variable `properties` was understood by two root classes: `Collection` and `GLMCompositePresentation`.

Class-Based Approach

While it is important for a developer to know the possible hierarchy to which the type of a variable may belong, it is sometimes also important to infer the precise class that represents the run-time type of the variable. Many of the analysed variables have an interface understood by many independent hierarchies, *i.e.,* hierarchies whose roots do not have a common superclass understanding the same interface, thus we wanted to verify how successfully heuristics would infer the precise class. A variable can have an interface understood by tens, hundreds, or even thousands of classes. Obviously, such information presented to a developer is not helpful. Hence we order them so that we promote the correct class (or classes) towards the top of the list.

Most of the existing simple type inference algorithms for dynamically-typed languages focus on the type hierarchy rather than on the precise

type.

The class-based approach takes into account all the possible classes inferred based on the variable's interface. This means that sorting classes is now applied on *sel_all_types*(v) rather than on *sel_types*(v). This indicates that *class_value_sel*(c) = *class_inst*(c) (or *name_occ*(c), depending on the heuristic) for any class c, since we are considering each class separately as a possible type.

In the example in Figure 4.3, the sets of possible types for the variable `properties` are

assign_types(`properties`) = {`OrderedCollection`}

sel_types(`properties`) = {`Collection`, `Bag`,
`SequenceableCollection`, `OrderedCollection`,
`GLMCompositePresentation`, `GTInspector`,
`GTMoldableDebugger`}

Let us emphasise here that no change is made to the set *assign_types*(v), but only to the set *sel_types*(v). We consider the set of explicitly assigned types to a variable to be truthful, as it is. The implications of this decision are discussed in Section 4.5.

4.4 Evaluation

We have implemented a proof-of-concept for Pharo Smalltalk.

In order to evaluate our assumptions, we have used five open-source projects, written in Pharo, for which we were able to collect run-time information that closely depicts their real usage: Roassal[4] [ABC+13], Glamour[5] [Bun09], Bloc[6], Morphic [FS07] and Moose[7] [NDG05, GÎ10, DGLD05, DLT00]. Roassal is an agile visualisation engine which graphically renders objects using short and expressive Smalltalk expressions (*i.e.,* an internal DSL). Glamour is a framework to describe the navigation flow of browsers. Bloc is a redesign of Morphic, a user unterface construction kit. Moose is a platform for software analysis. We wanted to avoid the use of unit tests, because there is no guarantee that they will reflect the

[4] `http://smalltalkhub.com/#!/~ObjectProfile/Roassal`
[5] `http://www.smalltalkhub.com/#!/~Moose/Glamour`
[6] `http://www.smalltalkhub.com/#!/~AlainPlantec/Bloc`
[7] `http://www.smalltalkhub.com/#!/~Moose/Moose`

complete picture of the project's usage. Four of these projects (Roassal, Glamour, Morphic and Bloc) provide a number of *example methods* that reflect the real usage of the corresponding project: Roassal has 948, Glamour 68, Morphic 29 and Bloc 202 of these methods. These methods are created by project developers to demonstrate the potential usage of the corresponding project. They serve as "main" methods and when executed provide an example of how the project may be employed. We ran them to record the run-time information about types of variables, *i.e.,* classes that represent the type of the object stored in the variable, during the execution of these methods. Run-time data for the Moose project was collected by actually performing software analysis on a couple of projects.

By instrumenting the source code of the projects to log the types of the variables, including instance variables, method and block arguments, and method and block temporary variables, and running these examples, we have recovered the run-time types of the variables. For this purpose we have used a mechanism to track the types of variables at run time, built on top of Reflectivity[8] [Den08], a tool used to annotate AST nodes with metalinks. We consider these types to be the actual, real types of the variables at run time, and hold those types to be ground truth to which results provided by the heuristics are compared.

The types of these variables are then inferred using the two heuristics. We have statically and dynamically collected enough information to infer the types of 5246 variables in these five projects and evaluated the results. This means that at least one message is sent to a variable, or at least one assignment of the newly created object to a variable is encountered in the source code during static analysis and that we had access to the run-time information about the types of these variables. Examples that we used to collect the run-time information about types covered 6009 variables in Roassal, 259 variables in Glamour, 1006 variables in Morphic, 220 variables in Bloc and 378 variables in Moose, thus in total 7872 variables. For 2626 variables, there was not enough information to infer types statically, *i.e.,* there was no assignment to the variable, nor any message sent.

We have measured the time needed to infer types for a variable, and the time needed to order possible types. The introduced overhead is 4.56% which we deem acceptable. The average time to infer and order types for a variable is 0.11 seconds, thus the approach remains fast and usable for

[8] `http://www.smalltalkhub.com/#!/~RMoD/Reflectivity`

program comprehension purpose.

The evaluation section is divided into three parts. First two part each focus on one of the used heuristics, and the last one presents a comparison with one existing approach to augment the precision of the same basic approach. We endeavour to answer the following research questions:

1. How successful is each of the heuristics?
2. For what kind of types, *e.g.,* library or project-related types, are the results correct?
3. How much does it improve the basic approach?

4.4.1 Class instantiation heuristic

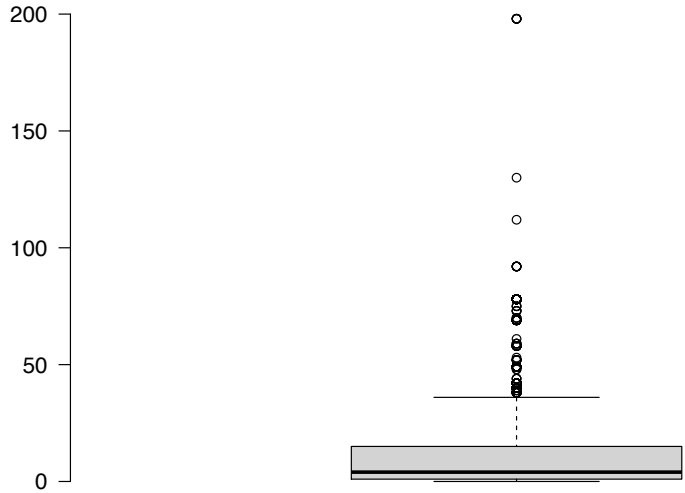

Figure 4.6: Distribution of the number of possible hierarchies for each statically analysed variable

Figure 4.6 shows the distribution of the number of possible hierarchies for each statically analysed variable. More than half of the variables (55%) have an interface understood by more than 4 independent hierarchies, and at

least 25% of the variables may have a run-time type belonging to more than 14 hierarchies. This is a consequence of cross-hierarchy polymorphism.

Project name	#of analysed variables	#of guessed variables	#of near-guessed variables	#of incorrectly-guessed variables	#of Object type
Roassal	4143	2584	557	620	382
Bloc	220	105	60	39	16
Glamour	136	84	6	12	34
Moose	150	68	8	44	30
Morphic	597	364	55	77	101
TOTAL	5246	3205 (61.09%)	686 (13.07%)	792 (15.09%)	563 (10.73%)

Distribution of guessed variables			
Project name	#of guessed variables	#of guessed variables - library type	#of guessed variables - package type
Roassal	2584	1429	1155
Bloc	105	78	27
Glamour	84	39	45
Moose	68	54	14
Morphic	364	193	171
TOTAL	3205	1793 (55.94%)	1412 (44.06%)

Correctly inferred types	
Basic algorithm	Class-instantiation heuristic
1120	3205

Table 4.2: Class instantiation heuristic — hierarchy-based approach

Hierarchy-based approach

If the class at the top of the list of sorted types is a superclass of the class that represents the actual run-time type of a variable, we label such a variable as *guessed.* If the variable has n run-time types, where $n > 1$, we consider it to be *guessed* if the first m classes of the statically inferred list include in their subhierarchies all run-time types of the variable. $m \geq 1$ is the size of the smallest set of statically inferred classes whose combined subhierarchies include all run-time types of the variable. We use the following example to explain the concepts used for the evaluation. If a variable has `Rectangle` and `Square` as run-time types, then $n = 2$. Let us suppose that the statically-inferred list of types for that variable includes in the first three positions, respectively, `Square`, `Rectangle` and `Triangle`. The first two classes include in their combined subhierarchies both run-time types, thus $m = 2$. Let us now suppose that the list of the statically-inferred types contains the same classes in the first three spots, but the first two classes in the reverse order, *i.e.,* it would be `Rectangle`, `Square` and `Triangle`. Since `Square` is a subclass of the `Rectangle` class, both run-time types would now be contained in the subhierarchy of the `Rectangle` class which is at the top of the statically-inferred types. Hence, in this case $m = 1$. We name this set of m classes *correct types.*

A summary of the evaluation results is given in Table 4.2. Results are presented per package, as well as in total. The results show that for 3205, *i.e.,* 61.09% variables, the heuristic is able to guess the hierarchy of possible types. Additional analysis revealed that the heuristic worked better for library types than for project-related types, with the corresponding percentages of 55.94% and 44.06% (table beneath in Table 4.2).

If the heuristic fails to guess the type of a variable, but the correct type is present in the top three classes, we call such a variable *near-guessed.* If the variable has n run-time types, where $n > 1$, we consider it as *near-guessed* if the set of first $m + 2$ classes of the statically inferred list of types include correct types. For example, if a variable has `Rectangle` and `Square` as run-time types (the same as in the example before), and the statically-inferred list of types includes in the first three positions, respectively, `Circle`, `Triangle` and `Parallelogram`, we consider it to be near-guessed. In this case $m = 1$, since hierarchy of the `Parallelogram` class includes the classes `Rectangle` and `Square`. The `Parallelogram` class is present at the third spot in the list, which is contained in the $m + 2$ first spots. If, instead, the statically-inferred list of types includes in the first

four spots `Circle`, `Triangle`, `Square` and `Rectangle`, respectively, we also consider it to be near-guessed. In this case $m = 2$ as two statically-inferred classes (`Square` and `Rectangle`, and in that order) are needed to cover all run-time types. For 686 variables, *i.e.,* 13.07%, correct run-time types were nearly-guessed, thus for 74.16% in total the heuristic offers reasonably sorted list of possible types.

In any other case, that is if the correct types would not be present in the first $m + 2$ spots in the list, or if the inference would fail to cover the correct types, the variable is defined as *incorrectly-guessed.*

To measure the improvement of the class-instantiation heuristic over the basic algorithm, we have compared the obtained results with the situations in which the basic algorithm was able to infer only the correct type, thus the results are unambigous. The comparison is presented as the last one in Table 4.2. These results show that the class-instantiation heuristic almost tripled the number of correctly inferred types.

Object type. As for the remaining 25.84% of variables, we have discovered that for almost a half, *i.e.,* 563 variables, the algorithm was not able to provide any information about the assigned types, and all the messages sent to these variables were defined in the `Object` class. Thus there was only enough static information to conclude the possible type as `Object`. We argue that these results could be discarded since they are easily identifiable and they do not provide any useful information to the developer. Beside 441 already defined methods in the `Object` class in Pharo, it is also possible with class extensions to add user-defined methods to the `Object` class. These messages can be sent to any Smalltalk object. Messages commonly sent to these 563 variables are:

- `rtValue:` which is specific to the Roassal project, but is defined as an extension method to the `Object` class and behaves in the same way as the message `value`, *i.e.,* returns the object to which the message is sent.

- comparison messages, like $=$, $==$, $!=$ *etc.*

- message `value` whose implementation in `Object` class returns the object to which the message is sent

- message `at:` which assumes that the receiver is indexable and answers the value of an indexable element of the receiver

Project name	#of analysed variables	#of guessed variables	#of near-guessed variables	#of incorrectly-guessed variables	#of Object type
Roassal	4143	2008	362	1391	382
Bloc	220	87	32	85	16
Glamour	136	52	5	45	34
Moose	150	45	17	58	30
Morphic	597	181	62	253	101
TOTAL	5246	2373 (45.23%)	478 (9.11%)	1832 (34.92%)	563 (10.73%)

Distribution of guessed variables			
Project name	#of guessed variables	#of guessed variables - library type	#of guessed variables - package type
Roassal	2008	963	1045
Bloc	87	67	20
Glamour	52	31	21
Moose	45	36	9
Morphic	181	136	45
TOTAL	2373	1233 (51.96%)	1140 (48.04%)

Table 4.3: Class instantiation heuristic — class-based approach

- messages which check whether the object is *null*, *e.g.,* `ifNil:`, `ifNotNil:` *etc.*

Class-based approach

As in the class-based approach we consider each of the possible types (both the recorded run-time types and the statically-inferred ones) as a class per se, that is without its subclasses, we emphasise that $n = m$ in this approach.

Results of the class-based approach are presented in Table 4.3. As

expected, the control- and data-flow insensitivity of the algorithm took their toll. The results show that in about 45.23% of cases, *i.e.,* for 2373 variables, by using this simple heuristic we can successfully infer the *correct classes* that represent the types of the variable. We emphasise that in this case the type inference heuristics are inferring the correct classes, since simple approaches developed for dynamically typed languages, *e.g., RoelTypes* [PMW09] and *EATI* [SLN14] infer solely the hierarchy of classes that represent the potential type of the variable. By analysis of the inferred types, we have established that this heuristic is working well both for project-related types, as well as for the types from the standard library, since 1123 of the correctly inferred types are library types, and 1140 are project-related ones.

For 478 additional variables, *i.e.,* 9.11% of variables, the heuristic failed to promote the correct types to the top of the list, but the correct types are present in the top $n + 2$ spots where n is the number of recorded run-time types. These are near-guessed variables.

Incorrectly-guessed types. For 1832 variables the correct types were not in the top $n + 2$ types in the list. For 26 variables the approach did not have enough information to conclude anything but types `UndefinedObject` or `Object`. The `UndefinedObject` class represents its sole instance, `nil`, used as a value for uninitialised variables. We argue that these types can be also discarded.

There are 115 incorrectly-guessed variables that have assigned types not corresponding to their run-time types. For example, one of the variables has the assigned object of type `OrderedCollection`, while the run-time types are `Array` and `OrderedCollection`. These two classes are commonly duck-typed in Smalltalk. Type `Array` is inferred based on the selectors sent to the variables, but was not present in the $n + 2$ spots in the list. Since both classes are subclasses of `SequenceableCollection`, the hierarchy-based approach is useful in this situation.

We have looked into the number of classes and hierarchies that understand the interface of these variables: at least half of the variables could have more than 301 different types. Most of them have an interface that is understood by multiple hierarchies. The number of these hierarchies ranges from 2 to 203 per variable, with a median of 11.

4.4.2 Name occurrence heuristic

Hierarchy-Based Approach

This heuristic works slightly better than the class instantiation heuristic. When we apply the hierarchy-based approach, we can see that for 3299 variables (62.88%), the heuristic is able to correctly infer the hierarchy of possible types for the variable. The corresponding results are presented in Table 4.4.

We again discard the variables for which we have not been able to infer any other type but `Object`. For 14.14% of variables, *i.e.,* 742 variables the correct hierarchy was not at the top of the list, but is present in the top $m + 2$ places, *i.e.,* they are near-guessed.

Again, we provide comparison with the basic algorithm (Table 4.4). The results are slightly better than the results of class instantiation heuristic.

Class-based approach

The results are similar to the previous heuristic *i.e.,* in 46.03% of cases (2415 variables) we succeeded to correctly infer the actual class that represents the type of the variable, as shown in Table 4.5. This heuristic also works well both for project-related types, and for the types from the standard library: 1236 of the correctly inferred types are library types, and 1179 are project-related types. Another 536 variables, *i.e.,* 10.22% of variables, are near-guessed.

Incorrectly-guessed types. There are 1732 (33.02%) additional variables for which the correct type was not in the top $n + 2$ types in the list.

Twenty six of these variables have an assigned type of `UndefinedObject`, and an interface consisting exclusively of messages defined in the class `Object`. Thus, we can only infer their types as sets of classes `UndefinedObject` and `Object`. These variables can be discarded.

Again, 114 of the incorrectly-guessed variables have been assigned in the source code multiple newly created objects whose types do not correspond to the actual run-time type of the variable.

Most of the remaining variables, *i.e.,* 1570 do not have any assigned type, hence inferring their type depends solely on their interface. Half of these variables have an interface understood by more than 323 classes and their type may belong to more than nine different class hierarchies.

Project name	#of analysed variables	#of guessed variables	#of near-guessed variables	#of incorrectly-guessed variables	#of Object type
Roassal	4143	2655	623	483	382
Bloc	220	114	55	35	16
Glamour	136	84	7	11	34
Moose	150	73	9	38	30
Morphic	597	373	48	75	101
TOTAL	5246	3299 (62.88%)	742 (14.14%)	642 (12.24%)	563 (10.73%)

Distribution of guessed variables			
Project name	#of guessed variables	#of guessed variables - library type	#of guessed variables - package type
Roassal	2655	1464	1191
Bloc	114	87	27
Glamour	84	40	44
Moose	73	57	16
Morphic	373	204	169
TOTAL	3299	1852 (56.14%)	1447 (43.86%)

Correctly inferred types	
Basic algorithm	name occurrence heuristic
1120	3299

Table 4.4: Name occurence heuristic — hierarchy-based approach

4.4.3 Comparison with EATI

Since the name occurrence heuristic has shown better results among the implemented heuristics, we compare it to EATI, a type inference technique that uses information available in the language ecosystem [SLN14], in contrast to approaches that use only information available in the project.

Project name	#of analysed variables	#of guessed variables	#of near-guessed variables	#of incorrectly-guessed variables	#of Object type
Roassal	4143	2026	423	1312	382
Bloc	220	95	31	78	16
Glamour	136	53	5	44	34
Moose	150	48	18	54	30
Morphic	597	193	59	244	101
TOTAL	5246	2415 (46.03%)	536 (10.22%)	1732 (33.02%)	563 (10.73%)

Distribution of guessed variables			
Project name	#of guessed variables	#of guessed variables - library type	#of guessed variables - package type
Roassal	2026	945	1081
Bloc	95	75	20
Glamour	53	32	21
Moose	48	38	10
Morphic	193	146	47
TOTAL	2415	1236 (51.18%)	1179 (48.82%)

Table 4.5: Name occurence heuristic — class-based approach

It has a simpler version of the fast type inference technique presented by Pluquet [PMW09] as the basis for its prototype implementation. It collects only messages sent directly to a variable, and not those sent through "getter" methods, and likewise for the assignments and "setter" methods. EATI gathers data about the frequency of message sends to instances of available types from the software ecosystem, and stores it in a central repository, for future queries. When possible types for a variable have been inferred, the likelihood of the variable being of the actual type "is computed based on how many times the messages sent to this variable have been observed to be sent to each potential type throughout the ecosystem" [SLN14].

Both RoelTyper [PMW09] and EATI [SLN14] employ a hierarchy-based approach for inferring types. That is why we compare it with the hierarchy-based approach. EATI showed a twofold improvement when compared to its basis, *i.e.,* RoelTyper, so we have decided to compare our results with the results produced by EATI.

Type inference	#of analysed variables	#of guessed variables	#of near-guessed variables	#of Object variables
Name occurrence heuristic — hierarchy-based	5246	3299	686	563
EATI	5246	2080	748	961

Table 4.6: Comparison with EATI

The results of our comparison are shown in Table 4.6. We have calculated the number of variables for which EATI succeeded to promote the correct type of the variable to the top of the list, and compared it with the number of variables for which name occurrence heuristic also succeeded to promote the correct type to the top of the list (#of guessed variables). We have also compared the number of variables for which both of the inference approaches failed to promote the correct type to the top of the list, but the correct type was among the first three on the list (number of near-guessed variables). On the same set of variables, the name occurrence heuristic performed significantly better: it correctly infers 58.6% more variables.

To be completely just, EATI does not employ getter and setter methods to obtain information about variable, nor does use it the heuristics listed in Table 4.1, thus this difference may be coming from there[9]. If we count also near-guessed variables together with guessed (since they can be considered as reasonable results to present to developer) we can see that there is an improvement of 40.9%. We deem these findings important, since our heuristic yields better results, and they are obtained with less effort and resources. In any case, EATI showed twofold improvement when com-

[9]By accident, this was not stated in the corresponding publication [MN16]

pared with the basic approach, while the heuristics presented here showed almost threefold improvement. Thus, we feel that the presented heuristics are certainly to some extent more precise than EATI. It should be noted also that EATI does not work well for the project-related types as it lacks awareness of these types, and focuses more on the types available broadly in the ecosystem, while our heuristics produce reasonable results both for project-related and library types.

4.5 Discussion and threats to validity

The main threat to validity comes from the run-time information we have used to evaluate our heuristics. We have chosen the projects Roassal, Glamour, Bloc, Morphic and Moose to evaluate the heuristics, since these projects benefit from sets of realistic example methods. We have chosen these examples over the unit tests, since we feel they illustrate the real usage of the projects, thus providing more insight into the actual software behaviour. It is an open question whether we have collected all possible run-time types for variables.

Another threat arises from the use of dynamic features and type predicates in Smalltalk. Dynamic features are seldom used in Smalltalk, but they are used to the extent that they must be taken into consideration [CRTR11], especially dynamic message send. Type predicates are prevalently used in Smalltalk [CRT+14]. In the manual investigation of our results, we have encountered variables queried for their type by sending the message `isKindOf:` and then being treated differently based on the answer. Since type inference heuristics presented in the paper are intended to be fast, they are flow-insensitive.

Our consideration of the lexical scope of an instance variable may have influenced the results. By all means, considering all the methods in the class that declares the instance variable, along with the methods of the subclasses, as the lexical scope of the variable may only improve the results.

We chose to treat the assignment types of a variable to be truthful as they are, without considering the subtypes. If the assigned type is `Collection`, we do not consider `Bag`, `OrderedCollection`, nor `SequenceableCollection` as the possible type of the variable (Figure 4.3). This choice is the same in both class-based and hierarchy-based approach. If we would consider the assigned types along with all subtypes

in class-hierarchy approach, the number of correctly inferred types may increase or decrease, due to the increased number of possible types for the variable. If the assigned type has a low value that pushes it towards the bottom of the list of inferred types, it increases the likelihood of it being outside of the set of the first $m + 2$ statically-inferred types.

We have used only intra-procedural analysis in our algorithm. Application of inter-procedural analysis would certainly improve the results.

While our first idea was to explore the class instantiation heuristic and name occurrence heuristic for type inference in dynamically typed languages, we have also tried to evaluate the heuristic that sorts the classes based on their number of live instances in the image. Since Pharo is a highly reflective and interactive IDE that supports live programming, many classes have live instances within the image, but this heuristic proved to be the least precise. The problem with this approach was that only 744 out of 8321 classes have live instances, compared to 4066 classes whose name is mentioned in the image, or 2906 classes that are instantiated somewhere in the image. This can also explain why the name occurrence heuristic performs slightly better than the class instantiation heuristic.

We intend to improve the class instantiation heuristic by analysing the ways a developer can create a new object of a class. In this implementation, we were only intercepting the messages that belong to any of the protocols `initialize`, `initialization` and `instance-creation`. Smalltalk is a highly reflective language, that includes dozens of ways to create a new object, and also allows a developer to define new ways by implementing class side methods.

One of the problems is also the set of available classes for every variable. Since Smalltalk is a dynamically typed language without a `main` method, determining the set of available classes for every variable requires control and flow analysis. We have tried to apply the approach used to guess the type of the method arguments in Smalltalk [SLN16], but the results were on average 30% worse than without applying this approach.

During our evaluation, we have statically analysed 9732 variables and collected run-time information to evaluate the types of 7872 variables. The intersection of these two sets yielded 5246 variables. The reason for this is that for 2626 variables for which we had run-time data there was not enough static information to infer their type, *i.e.,* no message had been sent to the variables, and they had no assigned type. On the other hand, for 4486 variables for which we had static information, there was no available dynamic information about their type. 49 of the variables have an interface

not understood by any single class in the image. We suppose that these 49 variables would only be inferable with a control-flow sensitive algorithm. 7099 of the analysed variables statically receive messages understood by more than one hierarchy. We think that these variables have the possibility of being duck-typed. This is a direct consequence of cross-hierarchy polymorphism. These results indicate a bigger presence of duck-typed message sends than the results of the study presented in Chapter 3 suggest. That is why we deem important for the future work to explore the actual usage of duck-typed variables, for these variables can be considered as a challenge to any type-inference algorithm.

4.6 Conclusion and future work

We have presented a couple of lightweight static heuristics that aim to provide precise type information by using simple algorithms. The implemented prototype for Pharo Smalltalk allows us to assess the proposed heuristics.

Our heuristics produced results comparable with the existing type inference algorithms, and tend to work quite well both for library and project-related types. While they can benefit from improvement, even in their simple form they provide us with promising results. The hierarchy-based approach is working better than class-based ones, which is to be expected. One reason for this is that more than 72.94% of analysed variables have an interface understood by more than one hierarchy.

During the analysis of the proposed heuristics, several opportunities for improvement have been remarked. With more complex approaches, it would be possible to identify the set of available classes for the project, so that the set of possible types of a variable can be shortened. Lexical similarities in between the variable name and class name may reveal the possible type of the variable. Type predicates used to ask the variable for its type can also give a hint about the possible type and improve heuristics.

5

Mining inline caches for class usage

5.1 Introduction

In the previous chapter, we have investigated the manner of improving the correctness of a flow-insensitive type inference algorithm by statically analysing the frequency of class usage and class instantiation in the source code. These heuristics focus on static rather than run-time data. Due to dynamic class loading, or use of reflection, static analysis may miss the use of certain types [LSS+15]. Hence, the usage of some classes may not even be visible at compile time, but only at run time. For this reason, run-time class usage information may be useful.

In order to speed up the execution in polymorphism presence, many virtual machines make use of Just-In-Time compilers that use inline caches [DS84, HCU91a]. Beside holding the information about methods previously executed at the message send, these caches also hold the information about receiver types. This information could be easily exploited in order to improve current tools for program comprehension. Inline caches have already been exploited for compiler optimisation pur-

poses [HCU91a, HU94], *i.e.,* type information has been fed back to the code, and in case of successful type checking at run time, the message send is inlined, and the code executes faster. However, to the best of our knowledge, it has still not been used to improve static type information for other message sends, for which the receiver type has not been collected from inline caches. We believe that this information collected during execution of any program written in the same language would add productively to the statically collected knowledge used for inferring a variable's type. As run-time information has been read from the virtual machine, no instrumentation is required.

We present an approach called *inline cache type inference* (ICTI) to exploit type information collected from inline caches during program runs from different systems written in the same language. We have again used *RoelTyper* [PMW09] for this purpose. Type information collected from inline caches is used to order statically inferred types of variables based on the class usage frequency during program runs. We propose that the frequency of class usage as the type of a receiver can serve as a reliable proxy to identify the type of a variable at run time. ICTI is not based on feeding the information collected from inline caches back to the message send for which type information has been collected, but to collect the information about the usage frequency of each class as a type of the receiver over time and use it as a proxy for the receiver type.

We have implemented a proof-of-concept for Pharo Smalltalk. We have used this implementation to evaluate our claim. The results show that the implemented heuristic is reasonably precise for more than 75% of the variables. The results of ICTI are compared to the unordered basic approach that we have used to construct a set of possible types for a variable [PMW09], and it more than doubled the number of correctly guessed types for a variable.

The rest of the chapter is organised as follows: Section 5.2 explains the virtual machine used for dynamic data collection. Next we define the used terminology and the implemented heuristic in Section 5.3. Section 5.4 shows results of the evaluation of the prototype. We then describe the potential threats to validity in Section 5.5 before concluding in Section 5.6.

5.2 Gathering of dynamic type information

Due to features, such as polymorphism, object-oriented languages represent a challenge for static optimization. Thus, modern virtual machines often rely on Just-in-Time (JIT) compilers which reason about receiver types based on the types met in the previous runs of the code [PVC01]. This design is used by Pharo Smalltalk virtual machine. We have built a run-time type gatherer using the infrastructure to extract types from previous runs of the JIT.

We explain briefly optimization of message send executions, and the process to extract receiver type information from the message sends.

5.2.1 Execution of message sends

By definition, in order to execute a method, a virtual machine interprets bytecode. When executing a message send, based on the type of the receiver and the message selector, the targeted method is looked for in the global look-up cache. If there is no cached method, the usual method look-up is performed.

One step further in the optimisation process is to store the look-up cache on the method level, rather than globally. This is performed by the baseline JIT compiler. If a method is frequently executed, it is translated directly to machine code, and the virtual machine will henceforth use the machine code version of the method. This version contains cache data for each message send separately, called an inline cache [DS84, HCU91b]. These caches hold information from previous method look-ups for the corresponding message send. As a side effect, they also contain receiver type information.

An inline cache can be in one of the following four states:

unused: when a method is translated for the first time, its inline caches are in the **unused** state. Around 30% of inline caches are always unused, because, for example, they belong to a never executed path in the method.

monomorphic: after a message send with unused inline caches is executed for the first time, the corresponding inline cache becomes **monomorphic**, *i.e.,* this cache holds one type of the receiver. 90% of inline caches are monomorphic.

polymorphic: when a monomorphic inline cache encounters a different receiver type than the one already registered, it becomes a **polymorphic** inline cache. 9% of all inline caches are polymorphic. A polymorphic inline cache contains up to six[1] different receiver types.

megamorphic: when an inline cache holds more than six different receiver types, it becomes **megamorphic**. 1% of all inline caches are megamorphic.

Information about receiver types may be extracted from monomorphic and polymorphic inline caches. Unused inline caches do not contain any receiver type information, and there is no reliable way to extract receiver type information from a megamorphic cache. The reason for this is of a technical nature. Machine code of all the eligible methods is contained in the machine code zone, memory of the fixed size contained in the virtual machine. When this memory hits its capacity (and it has usually size of up to 2MB), the virtual machine frees one quarter of the machine code zone that contains the least frequently executed methods. Since the zone is organised as a stack, it is then compacted to low addresses, in order for the JIT compiler to be able to put the new methods on the top of the machine code zone. All inline caches need to be relinked as the machine code method they refer to were potentially moved. Since megamorphic caches would have many addresses to relink, in order to avoid the performance issues, the virtual machine flushes them.

5.2.2 Run-time type information gatherer built

The Pharo virtual machine has in production a baseline JIT, thus it is possible through primitive methods[2] to extract receiver type information from the virtual machine, *i.e.,* from inline caches for methods that are translated to machine code.

Normally, from the virtual machine, it is possible to extract type information along with a bytecode program counter for a message send. Since the type inference algorithm is working on the AST level, we used a tool provided by a compiler [BDB+13], and usually employed by the debug-

[1] This number differs from one virtual machine to the other. In the Pharo virtual machine, it is set to six.

[2] Primitive methods are executed directly by the interpreter, and not by evaluating method statements. Some of the primitive methods have no other way to be executed.

ger [CGN14], to map the bytecode program counter to the corresponding AST node.

The run-time type gatherer is scheduled to run regularly (every second) in the Pharo image, in order to ensure that collected data is up-to-date. The gatherer queries the virtual machine for all the methods that have been recently executed, and collects receiver type information of the message sends within those methods.

5.3 Type inference algorithm

The algorithm and the terminology used are the same as in Chapter 4, explained in the Subsection 4.3.1, so we will not repeat it here. We will now explain the way to obtain *class_value*(c) for a class c.

5.3.1 Dynamic information

Let *MS* be the set of all message sends in the target programming language. Each message send has a receiver and a selector sent to the receiver.

$$\textit{run-time\_type}: \quad \textit{MS} \rightarrow \mathcal{P}(C) \tag{5.1}$$

$$\textit{class\_freq}(c) = |\{ms | c \in \textit{run-time\_type}(ms)\}| \tag{5.2}$$

$$\textit{class\_value}(c) = \sum_{\substack{c=sup^n(c'), n \in N \\ c' \in C}} \textit{class\_freq}(c') \tag{5.3}$$

Figure 5.1: Calculating class value.

Each class occurs as the type of a receiver for a message sent zero or more times (5.1). Based on the inline cache information collected during the image lifetime, we calculate the *class_freq* (5.2), as the number of message sends for which this class occurred as a receiver type during run time. *class_freq* is a global variable calculated per each class. Using this information we calculate *class_value*(c) for a class c, as the sum of *class_freq*(c') for each class c' which is a subclass of c (5.3). This information is used to sort the classes that represent possible types for a variable. We extract this information from the virtual machine, with the help of the

implemented run-time type information gatherer. Dynamically collected information is used to order separately two sets of classes: *assign_types*(v) and *sel_types*(v).

To present one list of possible types to a developer, we use *Assignment-FirstMerger* from the basic approach [PMW09]. This means that we give dominance to the assignment types rather than selector types. After sorting both lists of types, namely *assign_types*(v) and *sel_types*(v), we iterate through the list of *sel_types*(v) and remove all the classes that are related to any of the classes from *assign_types*(v), *i.e.,* are a superclass or a subclass of any of the assignment types. We append the remainder of the sorted list of selector types to the list of the assignment types.

Regarding the class-based approach, *class_value*(c) = *class_freq*(c) for any class c, since we are considering each class separately as a possible type.

5.4 Evaluation

5.4.1 Inline caching type gathering

In previous work type information from inline caches was fed back to the compiler for the purpose of optimisation [HCU91a, HU94]. To the best of our knowledge, this is the first experiment that explores to what extent run-time type information from the other packages is useful when trying to statically infer types in the packages separate from those whose run-time types are collected.

In order to collect run-time type information from inline caching we have run almost all the tests available in the Pharo image[3]. We ran 815 test classes, which left us with almost 12'000 test methods. These tests allowed us to collect the frequency of classes as message receiver types at run time for around 5'000 classes. The collected data has been used to calculate the class value, *class_value*(c) of each class c. We have measured the test execution times during data collection and compared them to the test execution without inline cache data collection. The average overhead introduced per test method is 0.6 milliseconds, *i.e.,* a bit less than 40%. The introduced overhead is acceptable, even though high, since we have run an unoptimised version of the type gatherer. We have run all the tests cases at once, and instructed the type gatherer to run every second, to collect

[3]Pharo 5.0 version 50761

enough data from inline caches. We believe that in reality the type gatherer can be instructed to run less frequently, thus the introduced overhead would be much smaller.

5.4.2 Projects used for evaluation

For the evaluation we have used four open-source Pharo projects that we used also throughout the evaluation in Section 4.4: Roassal, Glamour, Morphic and Moose. As before, we used example methods provided by Glamour, Morphic and Roassal projects, and for Moose we have collected run-time data by performing software analysis on the project.

Looking back at the inline cache type gathering phase, we run 815 tests that are part of the standard Pharo image. Note that two of the projects we have used for the evaluation, namely Glamour and Morphic, are part of the default Pharo image. During the inline cache type gathering phase we have omitted tests that belong to these projects. After removing the 76 tests belonging to these two projects from the 891 tests in the Pharo image, we were left with 815 tests.

We use run-time type information collected by running example methods to represent the ground truth in the evaluation phase, and compare them to statically inferred types. We do not collect any type information from these four projects during the inline cache type gathering phase.

Types of these variables are then inferred using ICTI. Examples that we used to collect the run-time information about types for which we were able to statically infer the type, *i.e.,* at least one message was sent to the variable, or there was an assignment of a newly instantiated type to the variable, covered 179 variables in Glamour, 257 variables in Moose, 1052 variables in Morphic and 3998 variables in Roassal. The numbers of variables per project are different from the previous section, since in between the two evaluation, new versions of the projects became available, which allowed us to increase the size of the set of variables used for evaluation.

5.4.3 Overall results — Hierarchy-Based approach

We repeat here the notion of *guessed* and *near-guessed* variables from Section 4.4.1.

If the class at the top of the list of sorted types is a superclass of the class that represents the actual run-time type of a variable, we label such a variable *guessed*. If the variable has n run-time types, where $n > 1$, we

Project name	#of analysed variables	#of guessed variables	#of near-guessed variables	#of incorrectly-guessed variables	#of Object type
Roassal	3998	2382	601	628	387
Glamour	179	103	25	13	38
Morphic	1052	684	174	52	142
Moose	257	155	33	33	36
SUM	5486	3324 (60%)	833 (15%)	726 (13%)	603 (11%)

Distribution of guessed variables			
Project name	#of guessed variables	#of guessed variables - library type	#of guessed variables - package type
Roassal	2382	1396	983
Glamour	103	47	55
Morphic	684	428	256
Moose	155	141	14
SUM	3324	2012 (60%)	1308 (40%)

Table 5.1: ICTI — hierarchy-based approach

consider it to be *guessed* if the first m classes of the statically inferred list include in their hierarchies all run-time types of the variable. $m \geq 1$ is the size of the smallest set of statically inferred classes whose combined hierarchies include all run-time types of the variable. We name this set of classes *correct types*.

If a variable is not guessed, but the correct type is present in the top three classes, we call such a variable *near-guessed*. If the variable has n run-time types, where $n > 1$, we consider it to be *near-guessed* if the set of first $m + 2$ types of the statically inferred list of types include correct types.

Overall results are presented in Table 5.1. Results are presented per package, as well as in total. The results show that the heuristic of ordering

Correctly inferred types	
Basic algorithm	ICTI
1453	3324

Table 5.2: Comparison with basic algorithm

types based on the frequency of a type being seen as the run-time type is able to guess the correct type in around 60% of cases. Deeper analysis revealed that, as expected, ICTI improved type inference of library types more than type inference of project-related types.

ICTI was additionally able to near-guess the type for about 15% of the variables, thus making the results optimal for 75% of variables in total.

As for the remaining set, we find 603 variables for which we were not able to infer any other type but `Object`. This means that no assignment was performed to the variable, nor any message other than messages defined in the `Object` class. Again, we argue that these results could be discarded since they are easily identifiable and they do not provide any useful information to the developer.

As we can see, the overall results are very similar to the results of the static heuristics from the previous chapter.

To be able to assess the improvement of ICTI compared to the basic algorithm, we have compared these results with the scenarios in which the basic algorithm infers nothing but the correct type of a variable. The comparison is presented in Table 5.2. These results show that ICTI more than doubled the number of correctly inferred types.

5.4.4 Difference between the basic algorithm and ICTI

The basic algorithm does not provide any ordering of the possible types of variables but presents them in random order. When it infers as possible types a set of classes that is a superset of the correct variable types, this information may lead to wasted developer effort and cause more harm than good. Promoting the correct type to the top of the list can be quite challenging without any flow analysis. We consequently investigate how large are the lists of possible types for variables for which the basic algorithm infers ambiguous results, and which were guessed by ICTI, *i.e.,* for 1871 variables. In Figure 5.2 we can see that around 20% of variables, *i.e.,* 375 variables out of 1871 have two statically inferred types. For the remaining

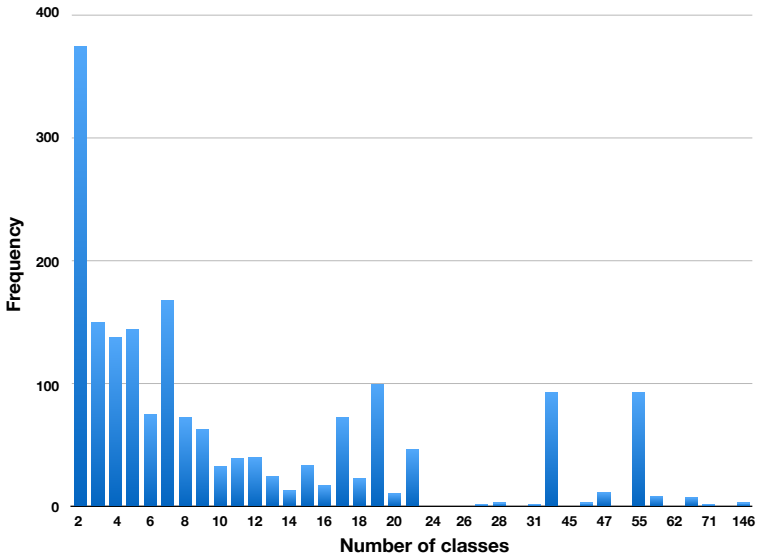

Figure 5.2: Frequency of the number of statically-inferred classes per variable

majority, the number of possible types is 3 or larger. If we remember that these types represent only hierarchy roots, the importance of promoting the right type to the top of the list is even higher. The median length of these lists is seven, and the maximum is 146, indicating that ICTI was able to promote the correct type to the top of the list even for lists larger than a hundred.

5.4.5 Not guessed variables

We further provide a deeper analysis of the reasons why for 726, *i.e.,* 13% of variables ICTI was not able to correctly infer types.

```
FAMIXNamedEntity>>                                                    1
   moosechefEqualsTo: anEntity moduloScope: aScope                    2
   | myselfRescoped |                                                 3
      ...                                                            4
      myselfRescoped isCollection                                    5
         ifTrue: [...]                                               6
         ifFalse: [...]                                              7
      ...                                                            8
```

Listing 6: Type predicate `isCollection` usage example

Run-time types are not inferable without control-flow sensitive analysis

A bit more than 6% of incorrectly guessed variables, *i.e.,* 46, have an interface that is not understood by any of their run-time types. Nine of these variables have an interface which is not understood by any class in the image. After closer investigation, we have discovered that these variables in most cases have an interface containing type predicates, *i.e.,* message sends that ask the variable for its type and program execution continues based on the provided answer. One of the most occurring examples is presented in the Listing 6 where the temporary variable `myselfRescoped` is queried for its type (line 5), and the execution flow is then divided according to whether it is a `Collection` or not. In this example, the type predicate `isCollection` is implemented in two classes: `Collection`, in which the method simply returns `true`, and `Object`, in which it returns `false`. This is the most common scenario, which indicates that these messages can be understood by any class.

Recent analysis [CRT+14] revealed that type predicates are used in almost all analysed projects in Smalltalk to perform an explicit type dispatch. Every 50th line of the code contains a type predicate. However, their analysis remains beyond the simple approach we have used in the basic algorithm.

Known duck-typed combinations

As already stated, *duck-typing* refers to the use of a variable to refer to objects of distinct classes that understand the same set of messages, without a common superclass understanding the same set. A classical ex-

ample of duck-typing in Smalltalk is the interchangeable usage of `Symbol`s and `BlockClosure`s, since instances of both classes understand the message `value:`. Beside `Symbol`s and `BlockClosure`s, the most commonly occurring pairs of classes are `Array` and `OrderedCollection`, and `CompiledMethod` and `ReflectiveMethod`. We have found 23 variables used to point to both of types included in a duck-typed pair. These variables demand flow-sensitive algorithms for their precise inference. Otherwise, the simple type inference algorithm may be improved for a specific language by including common duck-typed pairs into analysis.

Method and block arguments

More than half of the remaining set of incorrectly guessed variables, *i.e.,* 356 are method and block arguments. We do not go into deeper analysis but we propose a heuristic that combined with ICTI would improve the precision. The number of the possible types for these variables goes up to 124, with median of 18, thus being able to decrease the set of possible types would increase the likelihood of promoting the correct type to the top of the list.

A common practice in the Smalltalk community is to use method argument names to provide a hint at their expected type [GR83, Bec97]. For example, the method argument named `aString` suggests that the method expects an argument of type `String`. Type annotations in method argument names are also heavily used in other dynamically-typed languages like Python [XZC+16], Groovy [SF14] and Dart [FNT15]. Recent studies revealed that developers in Smalltalk do not practice this pattern consistently [SLN16], but it is common enough that it can be used as a heuristic. We have implemented the following heuristic:

1. first we extract the substring of the argument name starting from the first uppercase character to the end, *e.g.,* a variable named `aBlock` would yield `Block`

2. among classes that are inferred from a variable's interface, we would select only those classes whose name matches the regular expression `".*",` *extracted substring* `,".*"`

We would then proceed by ordering the remaining classes, and merging them with the assignment types. With this heuristic we are able to correctly infer types for the additional 110 out of 657 variables, thus for 17% of the

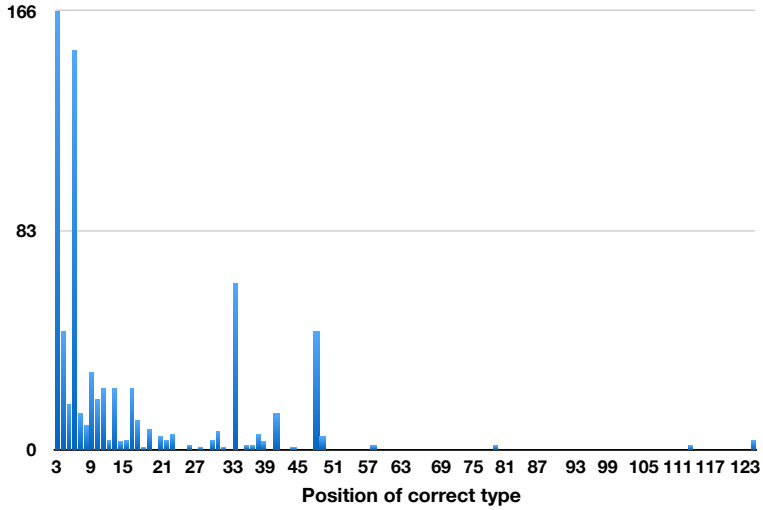

Figure 5.3: Position of correct type in the list

remaining set of incorrectly guessed variables we are now able to promote the correct type to the top of the list. These variables represent 31% of incorrectly guessed argument types.

5.4.6 Position of the correct type

We have investigated what is the distribution of k where k is the position of the correct type in the list of possible types. If we omit variables for which it is not possible to infer any other type than `Object`, the distribution of k reveals that in 85% of cases the correct type was present in the first three places of the list. For 3324 variables the correct type was present at the top of the list, while for 667 variables, the correct type was in the second place. We present the distribution of positions three and onward in Figure 5.3. For more than 90% of variables the correct type is present within the first ten inferred types.

Project name	#of analysed variables	#of guessed variables	#of near-guessed variables	#of incorrectly-guessed variables	#of Object type
Roassal	3'998	1529	320	1762	387
Glamour	179	71	17	53	38
Morphic	1052	345	105	460	142
Moose	257	97	28	96	36
SUM	5486	2042 (37%)	470 (9%)	2371 (43%)	603 (11%)

Distribution of guessed variables			
Project name	#of guessed variables	#of guessed variables - library type	#of guessed variables - package type
Roassal	1529	587	941
Glamour	71	33	38
Morphic	345	254	90
Moose	97	93	4
SUM	2042	967 (48%)	1073 (52%)

Table 5.3: ICTI — class-based approach

5.4.7 Overall results — Class-Based approach

Results of the class-based approach are presented in Table 5.3. As expected, control- and data-flow insensitivity of algorithm took their toll. ICTI was able to correctly infer the precise type of the variable in 37% of cases, and to near-guess it for only 9% more variables. These results are slightly worse than those presented for class-based approaches in Subsection 4.4.1 and Subsection 4.4.2.

5.5 Threats to validity

Beside threats to validity explained in the Section 4.5 we list here threats concerning the run-time type collection.

In order to implement ICTI for a dynamically-typed language, it suffices to be able to query the structure of the code in order to create a list of possible types for the variable, and to collect the type information from inline caches.

The main threat to validity comes from the collected run-time information from inline caches. If a class represents a variable's run-time type, but was not used during previous system runs, it would be missed, and results would not be correct. We have tried to address this problem by running all the tests in the image, to collect as much type information as possible.

There are a couple of drawbacks when it comes to the virtual machine.

Different object-oriented languages employ different performance optimisation techniques. Pharo Smalltalk uses a Just-In-Time compiler with inline caches. Beside JIT compilers, execution speedup may be achieved by employing a meta-tracing runtime compiler, *e.g.,* for languages like Lua and Python [Pal05, BCFR09]. Instead of storing the information about frequently invoked methods locally for a message send, these compilers store the information about linear sequences of methods that are frequently executed. Since they operate in a different manner, we cannot claim that our approach would work on top of them.

It may also happen that we do not collect all types encountered in inline caches. In practice, there is no infinite memory, thus only a subset of methods translated to machine code is accessible at a certain moment. The virtual machine contains a memory of a fixed size, named the machine code zone which contains the machine code of the most frequently executed methods, along with inline caches. When this zone is full, the garbage collector frees one quarter of it, removing the least frequently executed methods. Hence, the type information from these methods will be lost. To partially solve this problem, we have instructed the run-time type gatherer to run every second, and we have doubled the size of the machine code zone for our experiment. That is also why we deem that the introduced overhead would be much smaller in practice.

5.6 Conclusion

In this chapter we have presented a simple heuristic (ICTI) that aims to produce precise type information by using easily accessible information from inline caches. It collects the information about the frequency of class usage as receiver type from inline caches, and sorts statically inferred types based on this frequency. This way we try to compensate for the types that are used at run time, but are not visible statically in the code, due to reflection usage, or dynamic class loading. ICTI needs no instrumentation. It was evaluated using a prototype implemented in Pharo Smalltalk. We have focused our attention not only on inferring the root type of a variable, but also the correct subclass.

ICTI showed very similar results with the heuristics that use only statically collected information to order possible types for a variable. ICTI showed more than 100% improvement when compared with the basic approach.

6

Exploiting Type Hints in Method Argument Names

6.1 Introduction

In the previous two chapters we have explored heuristics used for the precision improvement of a simple flow-insensitive, intraprocedural approach that suffers from the problem of false positives, *i.e.,* classes that understand the interface of the variable, but do not represent its run-time type. Hence, it mostly *over-approximates* the results.

Other simple approaches perform interprocedural analysis, thus they are sensitive to calling relationships in between methods [Age95]. In any case, even these more complex static analyses tend to produce faulty results [LSS+15]. In the presence of reflection or dynamic class loading, these algorithms usually *under-approximate* the set of possible types for a variable, rather than over-approximating them [LSS+15]. This causes certain types to be missed, thus introducing the problem of *false negatives*, *i.e.,* the classes that are not inferred as possible types for a variable, yet represent the variable type at run time. A recent studies showed that developers in dynamically-typed languages often use dynamic and reflective

features [HH09, RLBV10, RHBV11, CRTR13], as well as in statically-typed languages [BSS+11]. For instance, a reflective method invocation is used in more than 60% of the analysed projects developed in Smalltalk, and for almost all of them it is not possible to discover the selector of the invoked method by only using simple static analyses [CRTR13].

In our experience, many developer communities strongly adhere to certain naming conventions. For example, a common idiom in dynamically-typed languages is to provide a type annotation for method arguments (*i.e.,* formal parameters to methods) [Bra15, Zan13, Bol10], *e.g.,* to name them after their expected type [Bec97]. Type hints in method arguments have a positive impact on program comprehension [SH14]. This is a motivation for encouraging developers to use type hints in method argument names.

These annotations are mainly intended to support the developer's reasoning about the variable, but they are also used by some development tools, *e.g.,* code completion [BDN+09]. The usage of identifier names has been explored to suggest a new identifier name [ABBS14, ABBS15], and study differences and similarities between formal method parameter and method argument names [LLS+16]. A recent study revealed that this annotating pattern is not strictly followed, but it may be successfully upgraded with simple heuristics for about a half of the explored method arguments [SLN16, XZC+16]. Groovy developers also quite heavily exercise type annotations in method argument names [SF14], as well as Dart developers [FNT15].

We believe that these hints can be of crucial importance for type inference in cases where the type of the variable cannot be statically inferred by traditional approaches. We propose a heuristic that combines an existing type inference algorithm with type hints from method argument names. Even though these hints may be used by any traditional type inference algorithm, we chose the Cartesian Product Algorithm (CPA) [Age95], whose precision heavily depends on the correctly inferred types for method arguments. CPA is a traditional type inference algorithm that was used as the base approach for several other algorithms [WS01, MSK07]. We employ an extension of the algorithm that is proposed by Spasojević *et al.* [SLN16] in order to obtain the type information from type hints in method argument names.

We have implemented a proof-of-concept prototype in Pharo Smalltalk and used this prototype to evaluate our hypothesis. When it comes to CPA, which depends on the type flow analysis, this heuristic showed significant improvement in the number of method arguments for which we were able

to correctly infer types, as compared with the types recorded at run time. In particular, the combination of CPA with type hints from method argument names, which we call CPA* is able to increase the size of the correct call graph by 30%, to analyse 52% more method arguments, and to correctly infer types of up to 81% more of the method arguments.

Structure of the chapter. First we motivate our approach in Section 6.2. We present the traditional algorithm that we used for the evaluation, and we explain the proposed heuristic in Section 6.3. We explain our prototype implementation in Section 6.4. Next we evaluate the approach in Section 6.5. We discuss possible threats to validity in Section 6.6 and conclude in Section 6.7.

6.2 Motivation

While type hints in identifiers are used in dynamically-typed languages, they are mostly intended to support human reasoning about the software at hand [BDN+09, SH14]. On the other hand, traditional type inference algorithms usually depend only on the analysis of language constructs, rather than on naming conventions. However, in the situations where the use of reflection is involved, these algorithms lack the information needed to infer types [LSS+15]. Further in this section, we emphasise the advantage of type hints for type inference through a real code example taken from Pharo Smalltalk[1].

```
ThreePhaseButtonMorph subclass:                                    1
    #PluggableThreePhaseButtonMorph
      instanceVariableNames: 'pressedImage target                  2
                        pressedImageSelector'                      3
      classVariableNames: ''                                       4
      package: 'Morphic-Widgets-Basic-Buttons'                     5
```

Listing 7: A subclass definition with three fields.

Consider lines 1-5 in Listing 7 that define a class named `PluggableThreePhaseButtonMorph`, a subclass of the class `ThreePhaseButtonMorph`. This class is a part of the package `Morphic`, a user interface construction kit [FS07] used for graphical representations in Pharo Smalltalk. Morphic is based on the idea that each object (a

[1] `http://www.smalltalkhub.com/#!/~Pharo/Pharo60/packages/Morphic-Widgets-Basic`

graphical component) is detachable from its parent, and can be manipulated on its own. The class `PluggableThreePhaseButtonMorph` allows the construction of a button that has three different images: one image for when the button is in the *on* state, another when it is in the *off* state and the third for when the button is just *pressed*.

Let us imagine that the developer wants to understand the control flow of the class, and how to manipulate one such morph. This class has a field named `pressedImage` (line 2 in Listing 7). In order to understand the implementation of the button, the developer wants to statically infer types of the expressions in the class. Since she does not want an over-approximation of the possible types, she decides to use the CPA. It closely tracks the flow of types from one expression to another, and propagates the types through connections between expressions. CPA needs an entry point — a *main* method—and for that purpose one of the factory methods can be used. These methods are defined on the *class side* of a class. Since everything in Smalltalk is considered to be an object, classes are objects, too. Thus the class side of a class defines methods that may be invoked on the class object. Pharo Smalltalk does not force a developer to write and use a main method in order to start a program execution. Any method in the project may be used as an entry point. The usual practice in Smalltalk includes using class side methods as main methods.

This factory method will first create an instance of the class `PluggableThreePhaseButtonMorph`, thus CPA will infer the type of this construction call to be of that class. The field `pressedImage` is not defined during object creation in the constructor, thus its value will be `nil`[2] at the beginning of the analysis.

```
1 PluggableThreePhaseButtonMorph>>#updatePressedImage
2    self pressedImage:(target perform:
3                               pressedImageSelector)
```

Listing 8: An example where static analysis cannot determine the type of a method argument.

CPA will continue to analyse the flow of method execution, and, concurrently, to infer the types of the expressions. During its evaluation, CPA will encounter the method named `updatePressedImage` in the class `PluggableThreePhaseButtonMorph`, presented in Listing 8. This

[2] `nil` is Smalltalk keyword for the undefined object, *i.e.,* equivalent to the **`null`** value in Java

method has one line of code, and it invokes the setter method for the field `pressedImage` (line 8 in Listing 8). The Smalltalk idiom is to name the setter method for an instance variable the same as the variable. The supposed value of the field `pressedImage` is the return value of the message send `target perform: pressedImageSelector`. Both `target` and `pressedImageSelector` are also fields of the same class.

The code `perform:` is a reflective way to invoke a method on the target object with the method name supplied as an argument to the method `perform:`. In this case a method with the name equal to the value of the variable `pressedImageSelector` is invoked on the variable `target`. Let us suppose that CPA was able to determine that the type of the variable `pressedImageSelector` is String. String is the expected type of the argument of the method `perform:`. However, even if CPA knows the type of `pressedImageSelector`, it does not know its actual value. Thus, the analysis is not able to compute which method will be invoked on the variable `target`. Recent studies revealed that invoking a method in this manner is quite common in Smalltalk code [CRTR13], as well as in other dynamically-typed languages [HH09, RHBV11] and for almost all of the occurrences, it is not easy to statically determine the actual value of the parameter.

In this case two scenarios are possible: either to assign the type `Object` to the return value of the message send `target perform: pressedImageSelector`, or to leave the set of inferred types for this expression empty. A common practice in these situations is to *under-approximate* the results, in the case when the concrete value cannot be determined by the analysis [LSS+15]. Otherwise, one type inferred as `Object` may heavily pollute the inference of other types in the system. This means that the set of possible types for the return value of the message send `target perform: pressedImageSelector` will be empty. Hence, following its next step, when the setter method `pressedImage:` (in the Listing 9) is entered, CPA will not know the type of its argument named `aForm`. The method implementation consists of one line of code in which the method argument is assigned to the field `pressedImage`. CPA works in such a way that it propagates all the inferred types of the right-hand side of an assignment to the variable on the left-hand side of the assignment. Since it did not infer type for the argument named `aForm`, subsequently, it will not be able to infer the type of the field `pressedImage`. Closer investigation of the class definition by the authors reveals that the field `pressedImage` is only assigned through this setter method, thus this

loss of information will highly impact further analysis.

```
4 PluggableThreePhaseButtonMorph>>#pressedImage: aForm
5    pressedImage := aForm.
```

Listing 9: The type hint from the argument name can help to detect the type of the method argument and subsequently the type of the field `pressedImage`.

We propose to exploit type hints in method argument names to highlight which classes are expected to represent the argument types at run time. We argue that these hints may improve analysis precision. In the example in Listing 9 the name of the method argument is `aForm`. A Smalltalk idiom is to embed a type hint in the method argument name, *i.e.,* to prefix the name of the expected class with the undefined article. For example, if the expected type is `String`, the corresponding argument name should be `aString`.

The analysis of method argument names embedded in Pharo from which the example is taken, will reveal that the expected type of the argument is represented by the class `Form`. It is an object used for holding images. Thus, with this small improvement, the analysis is able to assign the `Form` class to the set of possible types for the field `pressedImage`, and to continue performing the analysis with this information.

As a result, the developer will be issued with the information that the field `pressedImage` can have the class `Form` as its type. The actual type of the variable at run time is `ColorForm`, which is a subclass of `Form`. However, this is a common situation in object-oriented languages, due to a heavy use of polymorphism.

We argue that it is possible to introduce accurate information about the type of a variable with such a heuristic, and that this will provide more insightful information to developers for program comprehension.

6.3 Algorithm

6.3.1 The Cartesian Product Algorithm

The Cartesian Product Algorithm is a type inference algorithm developed by Agesen *et al.* [Age95]. It is implemented on top of the *Basic Type Inference Algorithm* [PS91] which statically models the run-time type flow.

It has been developed and implemented for Self [US87], a prototype-based programming language.

The program under analysis is depicted as a graph whose nodes represent program expressions, and directed edges portray run-time type flow. Each node holds the type information, *i.e.,* the set of classes, representing the possible types of the evaluation of the corresponding expression. For example, the node that represents a constructor call for the class `OrderedCollection` will hold as possible types the set with one element, namely the class `OrderedCollection`. If there is an assignment `x:=1` of value `1` to the variable named `x`, the node for this variable will have a class `Integer` as a possible type. This graph is depicted in Figure 6.1.

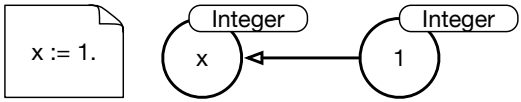

Figure 6.1: Type-flow graph example

Furthermore, if there is an assignment `y:=x`, the algorithm will propagate all the types from the node representing variable `x` to the node representing variable `y`. Hence, the variable node for variable `y` will also hold the class `Integer` as a possible type (Figure 6.2).

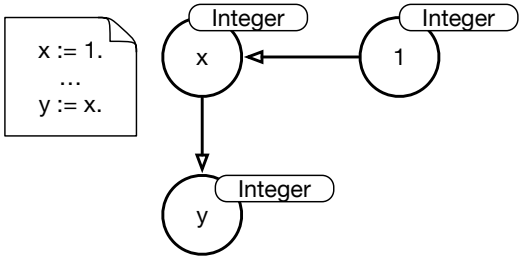

Figure 6.2: Type-flow graph example — continuation

If later in the code analysis variable `x` has a `String` object assigned to it, its node will contain henceforth two types: `Integer` and `String`. Since

there is a direct edge from the variable `x` to the variable `y`, the newly added type, that is `String`, will further be propagated to the node representing the variable `y`. This is depicted in Figure 6.3.

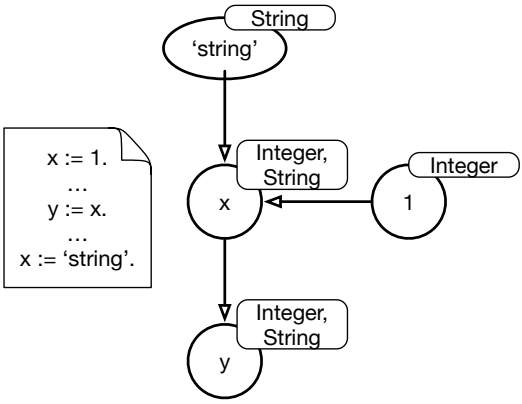

Figure 6.3: Type-flow graph example — propagation

The algorithm in the original paper [Age95] is explained for the Self language. Even though its implementation is language-independent, we will follow Agesen's terminology.

Let us denote by *E* the set of possible expressions in the project. As defined in Equation 6.1, it consists of literals (the set of literals is denoted by *L*), variables (*V*), blocks (*B)*, assignments, message sends and return statements. A block is a lexical closure [Pil04, BDN+09], present in many programming languages.

$$E = L \cup V \cup B \cup \{v := exp \mid v \in V, exp \in E\} \cup \{x\ msg{:}\ y | x, y \in E\} \cup \{\hat{}\, exp \mid exp \in E\} \tag{6.1}$$

The algorithm consists of three main steps:

1. create a node in the graph for each expression in the program, *e.g.,* variable named `x`, or an assignment `x:=1`

2. initially seed the types to the nodes for which the type can be determined before the analysis starts, *e.g.,* if there is an assignment `x:=1`, we can seed the node for variable `x` with the type `Integer`.

   The nodes that represent literals are seeded with the class that represents the type of the value held by a literal, *e.g.,* the node for the literal `'string'` is seeded with the type `String`.

3. propagate types along the edges between nodes, *e.g.,* if there is an assignment `y:=x`, the algorithm propagates all the possible types from the node representing variable `x` to the node representing variable `y`, thus `y` will also hold the type `Integer`

The algorithm infers types for expressions based on the constraints it creates during the analysis.

Assignment. During the analysis of an assignment expression `v:=exp` the algorithm infers the type for the expression `exp` on the right-hand side of the assignment, and constructs a directed edge from the node representing the expression `exp` to the node representing variable `v`, indicating that all the types inferred for `exp` should also belong to the set of possible types of variable `v`.

Message send. When it encounters a message send `x msg: y`, CPA creates edges from the arguments of the message send, to the formal parameters of the method that is supposed to be invoked, indicating a possible data flow at run time. If any of the argument expressions (including the receiver of the message send as the first argument) has more than one possible type, CPA creates a Cartesian product of the sets of possible types for the arguments, and analyses each of the combinations separately. Thus, whenever the algorithm enters a new method during the analysis, its formal parameters, and also the receiver, have *uniquely* identified types. After the method has been analysed, CPA caches the information about method argument types (including the type of the receiver of the message send) and return types. Thus, if at any time in the future the same method needs to be analysed with the same argument types, it can just collect the set of the return types from the cached information, without the need to analyse it again. That is how it preserves speed with accuracy [Age95].

The set of possibly invoked methods is constructed based on the possible types of the receiver of the message send. If there is more than one

possibly invoked method, edges from the message arguments are constructed to all of the corresponding formal method parameters, to ensure that the entire possible data flow is covered. For each method, the corresponding type of the message receiver is seeded as the value of `self`. Correspondingly, the type of the message send is the union of the return types of all methods that can be invoked at that message send.

Return statement. The return type of a method is the union of the types of all return statements `^exp`[3] within the method. The type of the return statement `^exp` is equal to the type of `exp`, the expression that constitutes the statement.

Block closure. A block is a lexical closure [Pil04, BDN+09], used to postpone the execution of the enclosing expressions. A block can access all the variables in the method in which it is defined, *i.e.,* method temporaries and fields of the enclosing class. It can also define its own arguments and temporaries. As for the message send, edges are constructed from the expressions supplied as arguments, to the corresponding block parameters. The return value of the block object is the type of the last expression in the block. Implementation issues concerning lexical closures are discussed in the original paper [Age95].

Since types are only propagated in the direction indicated by an edge, and never removed from the nodes, an inclusive relation, *e.g.,* `types(x)` $\subseteq$ `types(y)` will always hold for a node `x` with a direct edge to node `y`. Whenever a type is propagated along an edge, the propagation continues onward until it reaches the node that already contained that type, so no further propagation is required. This ensures that the analysis will eventually halt. Analysis continues until a fixed point is reached, and no more propagation is needed.

6.3.2 Type hints from method argument names

A common practice when writing dynamically-typed code is to name method arguments to provide a hint of the expected type. This practice differs from one language to another: while in Smalltalk the practice is to prefix the expected class name with an article, *e.g.,* `aString`, in Python

[3] Self syntax for the `return` statement is `^`.

the practice is to annotate a method argument in a specific manner[4], *e.g.,* `string: String` to indicate that a method argument named `string` expects an object of type `String`. As a consequence different manners of extracting the type hint from a variable name are needed for distinct languages: while in Smalltalk this involves parsing the method argument name, in Python this would require annotation analysis.

For this reason, we leave this part of the algorithm as abstract, supposing that in the specific language there is an implementation of a function returning the inferred type based on the analysis of the appropriate annotation of a method argument. We explain in detail the implementation of this function used for evaluation in Section 6.4.

6.3.3 Upgraded CPA — CPA*

We have implemented CPA with one additional step. Whenever CPA would encounter a method to analyse, just before creating the Cartesian product of the sets of possible types for the arguments, the new algorithm would infer method argument type(s) for each of the arguments, based on the type hints from the argument name. Then, for each argument, it would take the union of two sets: the set of inferred types by CPA, and the set of types inferred from the argument name. The algorithm would then continue as usual, to create a Cartesian product of the sets of types, and to analyse each combination separately. We call the new algorithm *CPA** .

6.4 Implementation

In this section we explain in detail the prototype implementation in Smalltalk.

6.4.1 CPA

In order to initially seed the types to the nodes, we have used a couple of heuristics to guess the type of the expression result assigned to the variable, as presented in the Table 6.1. The method for identity comparison (*i.e.,* ==) is implemented as a primitive method. Primitive methods are performed directly by the interpreter rather than by evaluating expressions in the method. Essential primitives cannot be performed in any other way. Thus,

[4]https://www.python.org/dev/peps/pep-0484/#acceptable-type-hints

Expression	Inferred type
$x = y \quad x == y \quad x \sim= y \quad x > y \quad x >= y$ $x < y \quad x <= y$	{True, False}
$x\ msg$, where x is a class and msg is any of the selectors from the set {new, new:, basicNew, basicNew:}	x

Table 6.1: Heuristics used to infer the type of the expression

there is no other way to handle these methods. Without the use of these heuristics, the analysis would lose precision. For example, if the algorithm were to analyse the method body, the inferred return type for method "==" would be the type of the object on which the method is invoked, rather than an instance of the `Boolean` class. The second row in the Table 6.1 refers to the primitive methods used to create new instances of a class.

6.4.2 Type Hints in Smalltalk

A recent study on the quality and usage prevalence of type hints from method argument names in Smalltalk [SLN16] revealed that type hints are provided only in around 36% of cases. If the expected type of a Smalltalk method argument is `String`, the corresponding argument should be named `aString`. The authors of the study proposed a couple of heuristics in order to improve the algorithm used for type hints. They managed to successfully guess the type for about half of the method arguments throughout the ecosystem. To guess the type of the argument named `arg`, we have used a slightly improved version of the algorithm proposed in this study. It is important to emphasise that the algorithm includes the following steps in the same order, and that the algorithm would proceed to the following step only if the previous step failed to provide a type:

1. if there is a class in the Pharo image with name matching `arg` (by ignoring upper and lower case differences), that class represents the type of the argument, *e.g.,* argument named `string` would have the type `String`

2. remove everything in the argument name before the first upper case letter, and match the rest with a class name, *e.g.,* an argument named `aString` would have the type `String`, but also an argument named `whateverString` would have the type `String`

3. if the argument name is `spec`, its supposed type is `MetacelloAbstractVersionConstructor`. `spec` is commonly used to name specifications of Metacello versions. Metacello is a package management system for Monticello, a version control system used for Smalltalk

4. if the argument name matches the regex "`.*(b|B)lock.*`", its type is `BlockClosure` (this class represents a lexical closure object in Smalltalk)

5. if the argument name matches the regex "`.*(o|O)rderedCollection.*`", its type is `OrderedCollection`

6. if the argument name matches the regex "`.*(a|A)rray.*`", its type is `Array`

7. if the argument name matches the regex "`.*(d|D)ictionary.*`", its type is `Dictionary`

8. if the argument name matches the regex "`.*(s|S)et.*`", its type is `Set`

9. if the argument name matches the regex "`.*(b|B)ag.*`", its type is `Bag`

10. if the argument name matches the regex "`.*(c|C)ollection.*`", its type is `Collection`

11. if the argument name matches the regex "`.*(s|S)tring.*`", its type is `String`

12. if the argument name matches the regex "`.*(s|S)ymbol.*`", its type is `Symbol`. Symbols in Smalltalk represent Strings that are created uniquely.

The algorithm in the study performed by Spasojević *et al.* did not include steps *5-9*. Thus, for any method argument whose name matches regular expression "`.*(c|C)ollection.*`" the inferred type would be `Collection`. However, since the `Collection` class is an abstract superclass of the classes `OrderedCollection`, `Array`, `Dictionary`, `Set` and `Bag`, and, based on our analysis of the current Pharo image, these subclasses are the most commonly used subclasses of the `Collection` class, we decided to treat them separately.

It is important for the algorithm to follow the steps in the indicated order, for the sake of the argument names like `aBlockAnsweringAString`, which is clearly a `BlockClosure` and not a `String`, or the argument named `aCollectionOfString` which is a `Collection` and not a `String` [SLN16].

If a method argument can expect an object of the type `BlockClosure` or `Symbol`, the convention is to name the argument *e.g.,* `aBlockOrSymbol`. The usual approach is to use the conjunction `Or` starting with a capital letter and followed by a capital letter, due to the Camel Case notation [WHH11]. While this is a convention in Smalltalk, a different approach might be taken in other languages [HBL+14]. In order not to lose type hints for these arguments, we would first split the argument name based on the appearances of the `Or` conjunction followed by an upper case, and then apply steps 1-12 from the algorithm on each of the substrings. The type(s) of the argument would be represented by the union of type(s) of each of the substrings.

Even though the presented algorithm is Smalltalk-specific, we believe that the same work can be performed in other dynamically-typed languages. The convention in Python is to suffix the method argument name with the expected type. For example, one would write `name: String` in a method definition that has an argument `name` whose expected type is `String`. In this case, the analysis of type hints would be even simpler than in Smalltalk, since it would not require regular expression matching, but just obtaining the string after the colon character. Recent studies revealed that type annotations are commonly used in Python [XZC+16], Dart [FNT15] and Groovy [SF14]. Hence, we believe that work similar to ours may be performed at least in these three languages.

6.5 Evaluation

For the evaluation we have used three open-source Pharo projects already presented in Section 4.4: Glamour[5] [Bun09], Roassal2[6] [ABC+13], and Morphic [FS07]. Each project provides a set of *example methods* that reflect their real usage: Glamour has 83 of these methods, Morphic 29, and Roassal 952. We have executed these methods, and the recorded runtime data serves as ground truth to which results provided by statical type

[5] `http://www.smalltalkhub.com/#!/~Moose/Glamour`
[6] `http://smalltalkhub.com/#!/~ObjectProfile/Roassal2`

Project name	# of methods	# of methods with arguments	# of arguments
Roassal	1371	483	745
Glamour	189	188	229
Morphic	677	675	935

Table 6.2: Run-time information

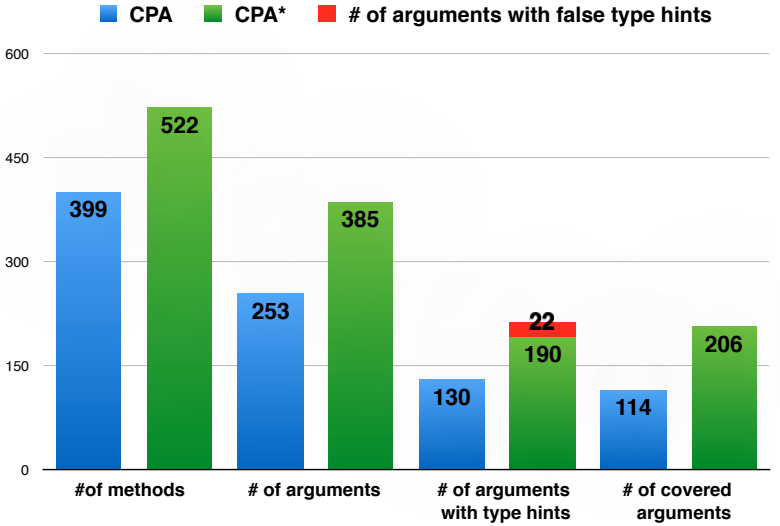

Figure 6.4: CPA and CPA* results

inference are compared.

As can be seen in Table 6.2, the execution of these examples covered in total 2237 methods from all three projects, out of which 1346 methods have at least one argument. We have recorded run-time information only for the methods in the packages, *i.e.,* we have excluded the library methods. We have used the same example methods as main methods, *i.e.,* the entry points to start the Cartesian product analysis.

The overall results may are represented in Figure 6.4. In total, CPA* analysed 30% more methods, which increased the number of analysed

Project name	# of methods	# of methods with arguments	# of arguments	# of arguments with type hints	# of covered arguments
Roassal	316	114	155	63	63
Glamour	62	62	75	48	41
Morphic	21	21	23	19	10
TOTAL	399	197	253	130	114

Table 6.3: CPA basic

arguments by 52%. Consequently, the number of method arguments that contain type hints was also increased by 63%. However, 22 out of 212 method arguments contained false type information, that is the type inferred from the argument name did not correspond to its run-time type. Yet, CPA* is able to correctly infer types for 81% more of the method arguments.

More detailed information regarding CPA and CPA* separately are presented in Table 6.3 and Table 6.4. When we executed the CPA analysis in its basic form, in our implementation it managed to cover 316 out of 1371 methods executed at run-time in the Roassal package, or one quarter of the actual call graph. As for the Glamour package, analysis covered 62 methods, *i.e.,* around 33% of the executed methods. We were surprised by the low number of statically analysed methods in the Morphic project. CPA managed to reach only 3% of the executed methods. A possible explanation is that 29 methods in the Morphic package use reflection, in which case traditional static analysis is helpless.

These 399 methods account for 253 method arguments. For 45% of these arguments, *i.e.,* for 114 out of 253, all types seen at run time were also inferred by CPA (this is presented in the column named "# of covered arguments" in Table 6.3). Hence, CPA did not underestimate types for a bit less than half of the arguments. The proportion of arguments with type hints in their name is quite different in all three packages: from 40% in Roassal to 83% in Morphic.

When we inferred types with the CPA* algorithm, we were able to cover 123 more methods, *i.e.,* 31% (Table 6.4). This indicates that 130 arguments in CPA analysis with type hints in their name traversed in CPA basic analysis managed to augment the size of the analysed call graph for around one method per argument with type hint. We deem this important,

Project name	# of methods	# of methods with arguments	# of arguments	# of arguments with type hints	# of covered arguments
Roassal	383	146	224	95	133
Glamour	66	66	80	50	46
Morphic	73	73	81	67	27
TOTAL	522	285	385	212	206

Project name	# of arguments with type hints	# of arguments with false type hints
Roassal	95	21
Glamour	50	0
Morphic	67	1
TOTAL	212	22

Table 6.4: CPA with type hints — CPA*

since it increases the size of the traversed call graph.

Accordingly, the number of analysed method arguments increased by 52%. The number of arguments for which all run-time types were also inferred by static analysis increased to 53.5%. The augmented algorithm therefore outperforms the basic algorithm, and correctly infers types for 81% more method arguments.

The column named "# of arguments with false type hints" in the table beneath in Table 6.4 provides information about misleading type hints. Most of them are due to the name `aCollection` for which only the class `Collection` can be inferred as type, and since `Collection` is an abstract class, it is not used as object type at run time. Also the argument name `aShape` is misleading, as the `Shape` class will be inferred as the possible type, although the run-time type is actually `RTShape` belonging to the Roassal package, which is not related to the `Shape` class even though it has a similar name.

There are a couple of situations where CPA* would infer both `BlockClosure` and `Symbol` as possible types for a method argument,

Project name	Time - CPA	Time - CPA and type hints
Roassal	11.5	12.1
Glamour	0.7	0.9
Morphic	1	1.6

Table 6.5: Time needed for the analysis in seconds

while at run time only one of them is recorded as the actual type. This would happen if a method argument is named *e.g.,* `aBlockOrASymbol`. Such variables often reflect a form of duck typing [TFH09]. In this specific case, for example, `Symbol` is duck-typed in Smalltalk to behave like a block in certain idiomatic scenarios. We therefore do not consider CPA* to be wrong just because only one of the two types is observed in practice. Also, if CPA* would infer as possible types for a method argument more classes than recorded at run-time, but the argument name clearly reveals that any of the inferred classes is expected at run time, *e.g.,* `aBlockOrANumber`, we do not count it as a false type hint. We choose to believe in the developer's suggestions, even though not all of the hinted classes are recorded during the execution.

This data shows that the combination of CPA with type hints from method argument names significantly increases the size of the analysed call graph, as well as the number of correctly inferred types for method arguments. This indicates that type hints from method argument names can improve CPA in dynamically-typed languages.

In order to evaluate whether or not this combination is still usable for the purpose of program comprehension, we have measured the time needed for both types of the analysis. On all three projects, we have found that the introduced overhead is quite divergent: from 5% to 60% (Table 6.5). The most overhead was introduced in the Morphic project, which is understandable, since the number of analysed methods more than tripled. This also indicates that Morphic contains method arguments with the largest number of type hints among the three analysed packages.

6.5.1 Argument names without inferred type hint

In all three projects, we have investigated the set of argument names for which the algorithm was not able to provide a hint. In this subsection, we

elaborate on our findings.

Roassal. The most frequently used names are, in order, `elements`, `objects`, `anElement`, `anObject`, `element`. Even though the analysis was not able to infer types from these names, a human can deduce that the first two arguments expect some kind of `Collection` object. Even though we think that this can be introduced as a rule, more research is needed to verify our assumption. As for the third name, there is no class named `Element`, but there is a class in the Roassal project named `RTElement`, so a developer may guess the expected type of the argument. The same applies for the argument named `element`. The name `anObject` is used as a method argument name in all three projects: in Roassal it is mostly used to indicate a member of a collection, or as an object on top of which a graphical representation is built; while in Glamour and Morphic it is mostly used as a name of a setter-method argument.

Glamour. Beside `anObject` the most common argument names for which the algorithm was not able to provide a type hint are `aPort`, `aPortReference`, `aPane` and `aPresentation`. These names correspond to types from the Glamour project, respectively, `GLMPort`, `GLMPortReference`, `GLMPane` and `GLMPresentation` (GLM stands for Glamour), which indicates that this kind of heuristic would greatly benefit from the input related to the project under analysis. Based on the examples from the Roassal and Glamour packages, the simple heuristic of removing the project-related prefix from class names and then inferring a type from the argument name would definitely improve the results. However, the improvement heavily depends on the practice of developers involved in the development of a certain project. In these particular projects, we assumed that the improvement would be significant, since about 30% of method arguments without type hints in Roassal and 50% of those in Glamour would provide a type hint by using this heuristic. Further investigation revealed that these classes are mostly abstract. If we have a bounded set of possible concrete classes in this case, inferring the actual type of such arguments would be feasible by applying an approach similar to the one we followed to detect the type for collections, as was explained in Subsection 6.3.2.

Morphic. An argument named `anImage`, which would suggest that an object of class `Image` is expected, actually expects an object of class `Form` which represents an array of pixels, used for holding images. No class `Image` exists in the version of Pharo used for the experiment. A method argument named `aFont` indicates that some kind of `font` is expected at

run time. While there is no class `Font`, there is an abstract class named `AbstractFont`, which defines the interface for fonts.

According to the findings in the Glamour project we suppose that giving precedence to the classes from the same package when analysing a method would improve the results. For example, a method argument `anAnnouncer` clearly indicates that it expects some kind of an announcer at run time. While the used algorithm for inferring the type from method argument name would conclude that its type is `Announcer`, in Glamour package its expected type is actually `GLMAnnouncer`, a subclass of the `Announcer` class.

6.6 Threats to validity

The first threat to validity comes from the run-time data we have used to evaluate the type inference. We have chosen the *Roassal2, Glamour* and *Morphic* projects for evaluation since we were able to run these projects in a way that closely resembles their real usage. However, it is an open question whether we have collected all possible run-time types for variables.

Another threat to validity comes from the quality of type hints in method argument names. While we have evaluated our heuristic in Smalltalk, exploring its performance in other dynamically-typed languages remains for the future work. Moreover, we have used the results provided by the study on a large set of Smalltalk projects, but it is an open question whether an arbitrary project will provide the same quality of type hints. As is presented in Section 6.5, some argument names have misleading type hints, while others provide a type hint for an abstract class, but it is obvious that a subclass will actually represent an argument type. Due to this problem, we have enriched the algorithm used to obtain a type from an argument name by the steps 5-9 (Subsection 6.3.2). The same kind of work is possible *e.g.,* for the argument name `aFont` (this name indicates an abstract class, while the usual run-time type is one represented by one of the subclasses), but we did not have any findings on the larger set of projects to support this.

The final threat comes from our choice of the basic algorithm. We chose CPA since its precision heavily depends on the correctly inferred types for method arguments and we think that it would greatly benefit from the proposed heuristic. However, we believe that this heuristic may be also combined with other simple type inference algorithms, for example Roel-

Typer [PMW09]. The heuristic is simple and fast enough not to endanger the speed of the underlying algorithm, yet we assume it would improve its precision.

6.7 Conclusion

Type annotations in method argument names are commonly used when coding in dynamically-typed languages. These annotations are intended for program comprehension purposes, but serve also as an input for different development tools.

On the other hand, inferring a variable type in dynamically-typed languages presents quite a challenge. In the presence of reflection and dynamic class loading, type inference algorithms lose on their precision, if they are data-flow insensitive.

We propose a heuristic that combines a traditional type inference algorithm that analyses language constructs, with type hints obtained from annotations of method arguments. We have performed a study to assess the possible impact of these hints on the results of a type inference algorithm called CPA. The obtained results are promising; the augmented algorithm outperforms the basic one. It increases the size of explored method arguments for a bit more than 50%, and subsequently correctly infers types for 81% more method arguments.

7 Conclusion

Static type information helps software maintenance, and eases program comprehension. Dynamically-typed languages decrease development time, but they lack static type information. Type inference algorithms may help developers to obtain the type information, but they need to be fast in order not to break work flow. For that matter, they need to remain simple, as complex analyses need more time and sources. However, the precision of the simple algorithms is hampered by polymorphism usage in object-oriented code, *i.e.,* they tend to produce false positives and negatives in their results.

In this thesis we have argued that the problem of false positives and negatives may be decreased by employing the lightweight heuristics to improve the precision of these algorithms, while preserving reasonable performance.

In order to assess the extent to which polymorphism is present in object-oriented languages, we first performed a large-scale study on more than one thousand projects, developed in both statically- and dynamically-typed code. While subtype polymorphism hampers program analysis in both types of languages, code analysis in dynamically-typed languages is additionally burdened by the use of cross-hierarchy polymorphism *i.e.,*

duck typing.

To mitigate the negative influence of polymorphism on type inference, we proposed four heuristics implemented on top of two simple type inference algorithms, fast enough to be usable during the development phase and not to break the developer's workflow. Each of these heuristics strives to employ distinct aspects of the code, and to handle different programming idioms. All of them achieved a significant improvement of precision when compared to the underlying algorithm.

In this chapter we summarise the contributions of this thesis, and we discuss the possible future directions of the research here presented.

7.1 Contributions

7.1.1 Large-scale polymorphism study

We presented a large-scale study of polymorphism presence both in statically- and dynamically-typed languages. We have found that subtype polymorphism is more widely present in dynamically-typed than in statically-typed languages, and that in dynamically-typed languages it is used in combination with cross-hierarchy polymorphism. Most code implements polymorphic selectors, and a polymorphic call site may have tens or even hundreds of method candidates. We have also found that duck typing is commonly used in dynamically-typed software, and that a vast majority of duck typed selectors has up to five methods introducing them, *i.e.,* being implemented in unrelated classes. Polymorphism is prevalent in object-oriented software. Along with the lack of static type information in dynamically-typed software, it heavily burdens program comprehension, and presents a significant challenge for type inference.

7.1.2 Lightweight heuristics

We presented four heuristics to improve the precision of simple type inference algorithms in the presence of polymorphic code.

Three of these heuristics are built on top of RoelTyper, which suffers to a great extent in the presence of subtype and cross-hierarchy polymorphism. In these situations, it tends to over-approximate the results. These heuristics showed a significant improvement over the basic algorithm, by outperforming it by more than 100%. All of the heuristics are cheap, easy to implement, and do not encumber the inference process in the sense of

introduced execution overhead. Chapter 4 presents two heuristics based only on statically collected information regarding class usage. Chapter 5 shows that dynamically-collected information from inline caches may also serve the same purpose. It takes precedence in the presence of reflective features, when it is not easy to determine the employed class by only static analysis. These heuristics touch different language idioms. All three of these heuristics showed similar results, with the improvement of the basic algorithm by more than 100%.

The fourth heuristic is implemented on top of CPA, which suffers in the presence of reflective and dynamic features, *i.e.,* it usually under-approximates the results. The heuristic exploits type annotations in method argument names and it significantly improves the power of the basic algorithm.

All of the heuristics substantially improved the performance of the underlying algorithm, with the introduction of acceptable overhead, which proves that they remain fast, and usable by a developer without breaking her workflow.

7.2 Future work and open questions

In this section we list future work directions and discuss our thoughts about open questions.

7.2.1 Choice of the basic algorithms

We chose RoelTyper and CPA, as they represent simple, yet efficient type inference algorithms that provide a developer with rapid information. While we find them representative for the field, there is an open question whether augmenting other type inference algorithms would yield different results.

Also, we have focused our attention on nominal types, as structural types burden developer reasoning [Str]. It may be that employing lightweight heuristics on algorithms working with structural types would yield different results.

7.2.2 Choice of the heuristics

Our idea was to start with the easiest heuristics to calculate, in order to retain the speed of the underlying algorithms. Our choice of the heuristics

fell on the four heuristics we have presented in this thesis, as they are simple enough not to complicate the type inference process. For instance, the heuristic that we employed on top of CPA provides results with only 10% overhead, and the other three heuristic introduce an overhead of maximum 5%. More complex heuristics, like restricting possible types only to the set of classes reachable from the code, may improve the results. It remains to be explored how much overhead they would introduce.

7.2.3 Combination of heuristics

Even though these heuristics are implemented and evaluated separately, they can be combined for further improvement of the results. For example, since statically collected class usage frequency information is lacking in the presence of reflection, it can be combined with ICTI to obtain run-time class usage information only for the situations when it cannot statically determine which class is used. Type hints from method argument names may also be employed to restrict the set of possible types for a method argument, when there is a clear hint of the expected type, and then to sort only the remaining classes.

7.2.4 Language idioms

Since we have implemented the four heuristics in Pharo Smalltalk, we accommodated some common Smalltalk coding idioms. For example, we considered as a constructor any method defined in any of the "`initialize`"-like protocols, as it is one of the Smalltalk coding styles.

We focus our attention neither on the information that may be extracted from the usage of dynamic and reflective features, nor on information that can be obtained from type predicates. The Smalltalk community uses heavily the mentioned features [CRTR13, CRT+14], and for a part of them it is statically possible to extract usable information. It may be that the additional focus on type predicates and reflective features would yield better results, possibly with negligible inference overhead.

7.2.5 Beyond Smalltalk

Pharo Smalltalk is a highly reflective object-oriented languages that allowed us to easily implement the heuristics without any additional requirements, but the Smalltalk knowledge. No additional tool was needed to

implement the proposed heuristics except Pharo itself. It may be more complex to deploy the analysis to other object-oriented languages, since we would need external tools to analyse the source code, *e.g.,* Moose.

We used a JIT compiler to obtained the information from inline caches. We cannot claim that this analysis is portable to some other kind of virtual machines, like those employing a meta-tracing runtime compile, *e.g.,* virtual machines for languages like Lua and Python [Pal05, BCFR09].

7.3 Summary

In this thesis we proposed several heuristics to mitigate the precision of simple type inference algorithms for dynamically-typed languages. Static type information has proven itself very useful during regular coding tasks, yet very hard to infer. While complex type inference algorithms may provide a developer with precise type information, they are time-consuming, hence impact the development workflow. Simple type inference algorithms provide rapid information, yet greatly suffer in the presence of polymorphic code.

We first performed a large-scale study about the presence of polymorphism in OO code to assess the criticality it poses on type inference. Our study revealed that polymorphism is omnipresent both in statically- and dynamically-typed code, and that the difference between two corpora is not large. Developers in dynamically-typed languages suffer significantly more from polymorphism than developers in statically-typed languages in regard to program comprehension. We believe that this is an indication of the importance of static type type information.

We then employed various heuristics to improve the precision of simple type inference algorithms, fast enough to be usable during regular coding tasks. Three of the heuristics mitigate the problem of false positives, and attempt to indicate the correct type of a variable among other inferred types. The fourth heuristic mitigates the problem of false negatives, *i.e.,* the problem of losing information in the presence of polymorphism and reflective features.

All of the implemented heuristics showed a significant improvement when compared to the underlying algorithm. We have also measured the time needed to obtain the type information, and found an overhead varying from 5% to 10%, which we deem acceptable.

Appendices

A

Implementation and Usage of the Type Inference Tool in Pharo

In this section we provide an instruction of how to download and use the type inference tools explained in the thesis. The corresponding information can be found on `http://smalltalkhub.com/#!/~NevenaMilojkovic/type-inference-heuristics` and `http://smalltalkhub.com/#!/~NevenaMilojkovic/CPA`.

A.0.1 Tool for ordering classes based on the heuristics presented in Chapter 4 and Chapter 5

The following Gofer script should be executed in Pharo (`http://pharo.org/`):

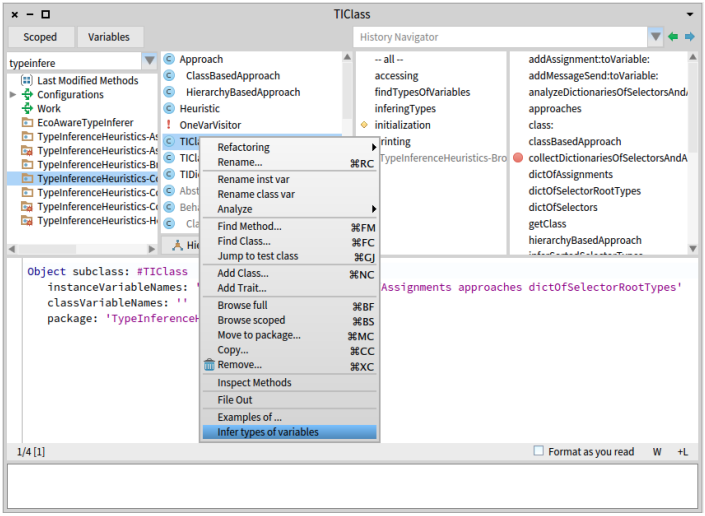

Figure A.1: Inferring types of variables in the `TIClass` class

```
Gofer new
   url: 'http://smalltalkhub.com/mc/',
         'NevenaMilojkovic/',
         'type-inference-heuristics/main';
   package: 'ConfigurationOfTypeInferenceHeuristics';
   load.
(Smalltalk at:
   #ConfigurationOfTypeInferenceHeuristics)
   loadDevelopment.
```

The execution of the script will take up to a couple of minutes, in order to initialise the information necessary for the tool.

In order to infer types of variables defined within a class one can write the following code in the Playground (`TIClass fromClass:` *class − object*) `inferTypes` and `do-it`. The argument of the method `TIClass class>>#fromClass:` should be a class object.

Another way to infer types of variables defined within a class is to open the class menu (by right-clicking on the class name in Nautilus) and choose the option `Infer types of variables` (Figure A.1).

The execution of this message will result in a browser with the list

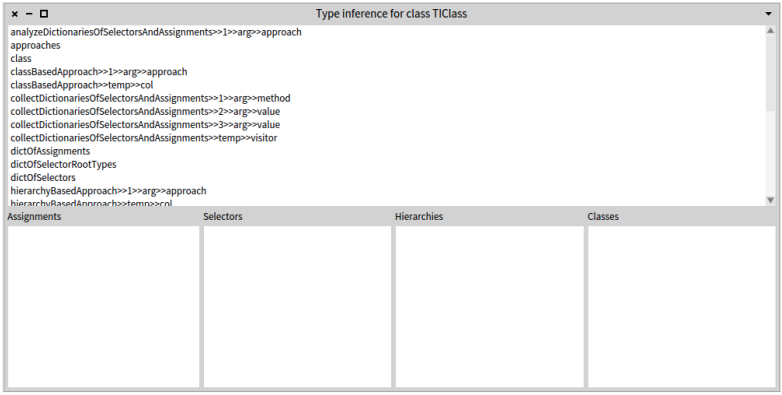

Figure A.2: Inferred types of variables in the `TIClass` class

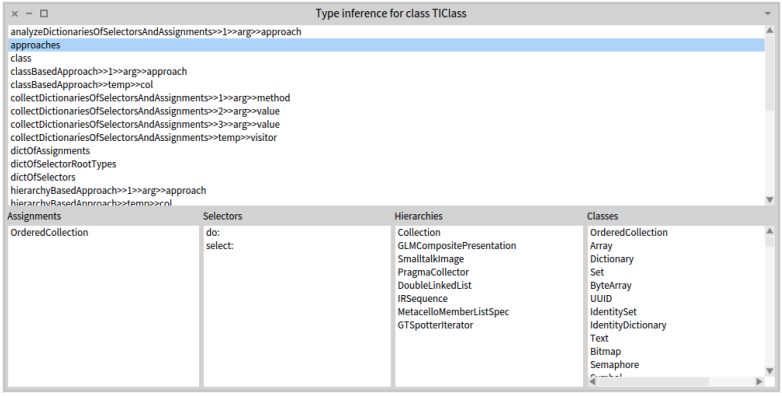

Figure A.3: Types of the selected variable `approaches`

of all the variables contained within the class, ordered alphabetically by their names (Figure A.2). Instance variables are presented solely by their name, while method arguments and temporaries are presented in the form `method_name>>arg>>var_name` for an argument or `method_name>>temp>>var_name` for a temporary variable. Block arguments and variables are presented in a similar form, with the method name being followed by the ordinal number of that block within the method body. For example, the name of the variable `classBasedApproach>>1>>arg>>approach` within the class `TIClass` indicates that the variable `approach` is an argument of the first block to be *opened* in the method `TIClass>>#classBasedApproach`. There is an emphasise of the word *opened* since the tool is assigning the ordinal numbers to the blocks inside the method body. The numbers are assigned based on the order of blocks being opened within the method body.

The bottom part of the browser contains four parts: `Assignments`, `Selectors`, `Hierarchies` and `Classes`. When a variable is being selected from the upper rectangle (Figure A.3), these fours parts will contain the following information, respectively:

- list of all the classes assigned to the variable
- list of all the messages being sent to the variable
- list of all the hierarchy roots to which the variable's type may belong to
- list of all the classes which may represent the variable's type

Three of these lists, *i.e.,* `Assignments`, `Heirarchies` and `Classes` lists, contain type information that is ordered based on the likelihood of the class being correct.

An object of type `RBVariableNode` may also be asked for its types by sending it a message `inferTypes`. The result is a collection of the inferred types ordered based on the likelihood of being correct. Alternatively, within the source code, one can select a variable of interest, and choose the option `Infer types` from the menu obtained by right click (Figure A.4). The result is also a collection of the ordered inferred types.

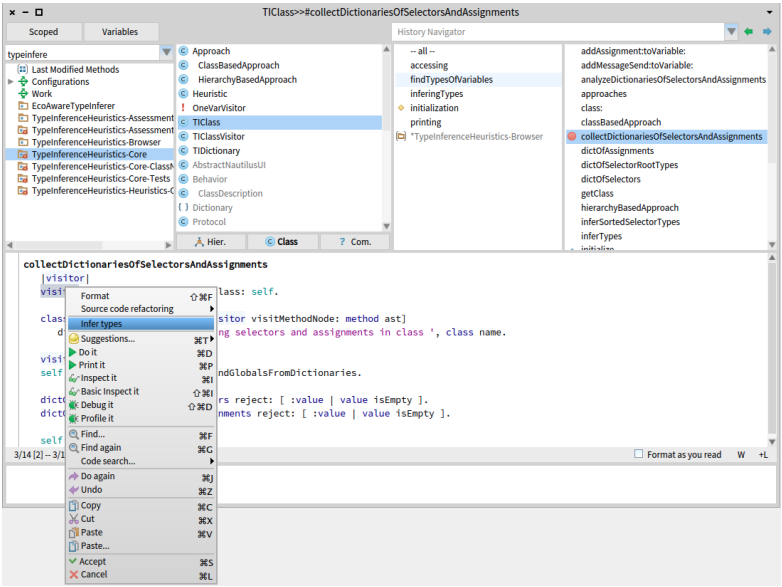

Figure A.4: How to infer types of the selected variable `visitor`

A.1 Assessment of the heuristics presented in Chapter 4 and Chapter 5

In order to assess the presented heuristics, the following Gofer script should be executed in Pharo:

```
Gofer new
   url: 'http://smalltalkhub.com/mc/',
         'NevenaMilojkovic/',
         'type-inference-heuristics/main';
   package: 'ConfigurationOf',
            'TypeInferenceHeuristicsAssessment';
   load.
(Smalltalk at:
   #ConfigurationOfTypeInferenceHeuristicsAssessment)
   loadDevelopment.
```

This will load all the classes needed for the assessment, and analyse the

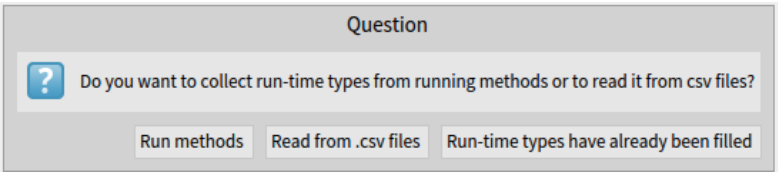

Figure A.5: Collecting run-time types

image in order to collect the necessary information about class frequency usage within the current Pharo image. This may take up a couple of minutes. The corresponding progress is presented in the left upper corner of the image.

After the setup, the analysis should be run by executing the following code `TIAssessment assessAllHeuristics`. The user will first be asked whether the system should run the example methods from the packages used for the evaluation (Roassal2, Glamour, Morphic and Moose) or if the run-time types can be read from the corresponding csv files (files used for the evaluation can be downloaded from `https://github.com/NevenaMilojkovic/type-inf-heuristics/tree/master/csv-files`), as presented in Figure A.5. The user should put these files in the repository `FileSystem workingDirectory/'runTimeTypes'`. After collecting the necessary information to infer types for the variables in the corresponding packages (which may take several minutes), the system will ask the user to select the heuristics based on which she would like to order possible types (Figure A.6).

After this step, the system will continue its analysis. The results of the analysis will be stored in the repository `FileSystem workingDirectory/results` in the csv format. Each csv file corresponds to one analysed project.

A.2 Assessment of CPA*

In order to assess CPA* , the following Gofer script should be executed in Pharo:

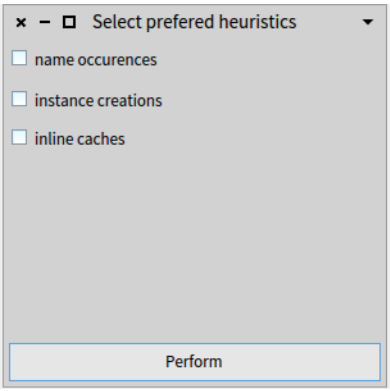

Figure A.6: Selecting the preferred heuristics

```
Gofer new
   url: 'http://smalltalkhub.com/mc/',
         'NevenaMilojkovic/CPA/main';
   package: 'ConfigurationOfCPA';
   load.
(Smalltalk at: #ConfigurationOfCPA) loadDevelopment.
```

In order to analyse the projects Roasssa2, Glamour and Morphic, one should execute the code `CPAEvaluation analyse`.

In order to assess these results, files from the link `https://github.com/NevenaMilojkovic/type-inf-heuristics/tree/master/CPARunTime` should be stored inside the repository `FileSystem workingDirectory/'CPARunTime'`. These files contain recorded run-time information for the variables defined in the mentioned projects. They are imported into image by executing `CPAEvaluation importRunTimeTypes`.

In order to evaluate CPA* , one should execute the code `CPAEvaluation assess`. This execution will create a file named *CPAResults.txt* inside the repository `FileSystem workingDirectory/'CPAResults'`. This text file contains the information about the analysed methods and their arguments.

Bibliography

[ABBS14] Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles Sutton. Learning natural coding conventions. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 281–293, New York, NY, USA, 2014. ACM.

[ABBS15] Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles Sutton. Suggesting accurate method and class names. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 38–49, New York, NY, USA, 2015. ACM.

[ABC+13] Vanessa Peña Araya, Alexandre Bergel, Damien Cassou, Stéphane Ducasse, and Jannik Laval. Agile visualization with Roassal. In *Deep Into Pharo*, pages 209–239. Square Bracket Associates, September 2013.

[ACF+13] Esteban Allende, Oscar Callaú, Johan Fabry, Éric Tanter, and Marcus Denker. Gradual typing for Smalltalk. *Science of Computer Programming*, August 2013.

[ACFH11] David An, Avik Chaudhuri, Jeffrey Foster, and Michael Hicks. Dynamic inference of static types for Ruby. In *Proceedings of the 38th ACM Symposium on Principles of Programming Languages (POPL'11)*, pages 459–472. ACM, 2011.

[AGD05] Christopher Anderson, Paola Giannini, and Sophia Drossopoulou. Towards type inference for JavaScript. In *Proceedings of the 19th European Conference on Object-Oriented Programming*, ECOOP'05, pages 428–452, Berlin, Heidelberg, 2005. Springer-Verlag.

[Age95] Ole Agesen. The Cartesian product algorithm. In W. Olthoff, editor, *Proceedings ECOOP '95*, volume 952 of *LNCS*, pages 2–26, Aarhus, Denmark, August 1995. Springer-Verlag.

[AH95] Ole Agesen and Urs Hölzle. Type feedback vs. concrete type inference: A comparison of optimization techniques for object-oriented languages. In *Proceedings of the Tenth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, OOPSLA '95, pages 91–107, New York, NY, USA, 1995. ACM.

[AH96] Gerald Aigner and Urs Hölzle. Eliminating virtual function calls in C++ programs. In P. Cointe, editor, *Proceedings ECOOP '96*, volume 1098 of *LNCS*, pages 142–166, Linz, Austria, July 1996. Springer-Verlag.

[AM91] Alex Aiken and Brian Murphy. Static type inference in a dynamically typed language. In *Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '91, pages 279–290, New York, NY, USA, 1991. ACM.

[ANMM06] Chris Andreae, James Noble, Shane Markstrum, and Todd Millstein. A framework for implementing pluggable type systems. In *OOPSLA '06: Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 57–74, New York, NY, USA, 2006. ACM Press.

[APS93] Ole Agesen, Jens Palsberg, and Michael I. Schwartzbach. Type inference of SELF: Analysis of objects with dynamic and multiple inheritance. In Oscar Nierstrasz, editor, *Proceedings ECOOP '93*, volume 707 of *LNCS*, pages 247–267, Kaiserslautern, Germany, July 1993. Springer-Verlag.

[ÅW15] Beatrice Åkerblom and Tobias Wrigstad. Measuring polymorphism in Python programs. In *Proceedings of the 11th Symposium on Dynamic Languages*, DLS 2015, pages 114–128, New York, NY, USA, 2015. ACM.

[BAT14] Gavin Bierman, Martìn Abadi, and Mads Torgersen. Understanding TypeScript. In *ECOOP 2014 – Object-Oriented*

Programming: 28th European Conference, Uppsala, Sweden, July 28 – August 1, 2014. Proceedings, pages 257–281, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[BCFR09] Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski, and Armin Rigo. Tracing the meta-level: PyPy's tracing JIT compiler. In *ICOOOLPS '09: Proceedings of the 4th Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, pages 18–25, New York, NY, USA, 2009. ACM.

[BDB+13] Clément Bera, Stéphane Ducasse, Alexandre Bergel, Damien Cassou, and Jannik Laval. Handling exceptions. In *Deep Into Pharo*, page 38. Square Bracket Associates, September 2013.

[BDN+09] Andrew Black, Stéphane Ducasse, Oscar Nierstrasz, Damien Pollet, Damien Cassou, and Marcus Denker. *Pharo by Example*. Square Bracket Associates, 2009.

[Bec97] Kent Beck. *Smalltalk Best Practice Patterns*. Prentice-Hall, 1997.

[BG93] Gilad Bracha and David Griswold. Strongtalk: Typechecking Smalltalk in a production environment. In *Proceedings OOPSLA '93, ACM SIGPLAN Notices*, volume 28, pages 215–230, October 1993.

[Bol10] M. Bolin. *Closure: The Definitive Guide: Google Tools to Add Power to Your JavaScript*. O'Reilly Media, 2010.

[Bra04] Gilad Bracha. Pluggable type systems. In *In OOPSLA'04 Workshop on Revival of Dynamic Languages*, October 2004.

[Bra15] Gilad Bracha. *The Dart Programming Language*. Addison-Wesley Professional, 1st edition, 2015.

[BS96] David F. Bacon and Peter F. Sweeney. Fast static analysis of C++ virtual function calls. *SIGPLAN Not.*, 31(10):324–341, October 1996.

[BSS+11] E. Bodden, A. Sewe, J. Sinschek, H. Oueslati, and M. Mezini. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In *2011 33rd International Conference on Software Engineering (ICSE)*, pages 241–250, 2011.

[Bun09] Philipp Bunge. Scripting browsers with Glamour. Master's thesis, University of Bern, April 2009.

[BWDP00] Lionel C. Briand, Jürgen Wüst, John W. Daly, and D. Victor Porter. Exploring the relationships between design measures and software quality in object-oriented systems. *Journal of Systems and Software*, 51(3):245–273, 2000.

[Car98] Michelle Cartwright. An empirical view of inheritance. *Information and Software Technology*, 1998.

[CCSL14] Andrea Caracciolo, Andrei Chiş, Boris Spasojević, and Mircea Lungu. Pangea: A workbench for statically analyzing multi-language software corpora. In *Source Code Analysis and Manipulation (SCAM), 2014 IEEE 14th International Working Conference on*, pages 71–76. IEEE, September 2014.

[CF91] Robert Cartwright and Mike Fagan. Soft typing. In *PLDI '91: Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, pages 278–292, New York, NY, USA, 1991. ACM.

[CGN14] Andrei Chiş, Tudor Gîrba, and Oscar Nierstrasz. The Moldable Debugger: A framework for developing domain-specific debuggers. In Benoît Combemale, David J. Pearce, Olivier Barais, and Jurgen J. Vinju, editors, *Software Language Engineering*, volume 8706 of *Lecture Notes in Computer Science*, pages 102–121. Springer International Publishing, 2014.

[CRT+14] Oscar Callaú, Romain Robbes, Éric Tanter, David Röthlisberger, and Alexandre Bergel. On the use of type predicates in object-oriented software: The case of Smalltalk. In *Proceedings of the 10th ACM Dynamic Languages Symposium (DLS 2014)*, pages 135–146, Portland, OR, USA, 2014. ACM Press.

[CRTR11] Oscar Callaú, Romain Robbes, Éric Tanter, and David Röthlisberger. How developers use the dynamic features of programming languages: The case of Smalltalk. In *Proceedings of the 8th Working Conference on Mining Software Repositories (MSR 2011)*, pages 23–32, New York, NY, USA, 2011. IEEE Computer Society.

[CRTR13] Oscar Callaú, Romain Robbes, Éric Tanter, and David Röthlisberger. How (and why) developers use the dynamic features of programming languages: the case of Smalltalk. *Empirical Software Engineering*, 2013.

[DBM+96] John Daly, Andrew Brooks, James Miller, Marc Roper, and Murray Wood. Evaluating inheritance depth on the maintainability of object-oriented software. *Empirical Software Engineering*, 1(2):109–132, 1996.

[DDM+03] Serge Demeyer, Stéphane Ducasse, Kim Mens, Adrian Trifu, and Rajesh Vasa. Report of the ECOOP'03 workshop on object-oriented reengineering. In *Object-Oriented Technology (ECOOP'03 Workshop Reader)*, LNCS, pages 72–85. Springer-Verlag, 2003.

[Den08] Marcus Denker. *Sub-method Structural and Behavioral Reflection*. PhD thesis, University of Bern, May 2008.

[DGC95] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In W. Olthoff, editor, *Proceedings ECOOP '95*, volume 952 of *LNCS*, pages 77–101, Aarhus, Denmark, August 1995. Springer-Verlag.

[DGLD05] Stéphane Ducasse, Tudor Gîrba, Michele Lanza, and Serge Demeyer. Moose: a collaborative and extensible reengineering environment. In *Tools for Software Maintenance and Reengineering*, RCOST / Software Technology Series, pages 55–71. Franco Angeli, Milano, 2005.

[DGN05] Stéphane Ducasse, Tudor Gîrba, and Oscar Nierstrasz. Moose: an agile reengineering environment. In *Proceedings of ESEC/FSE 2005*, pages 99–102, September 2005. Tool demo.

[DLT00] Stéphane Ducasse, Michele Lanza, and Sander Tichelaar. Moose: an extensible language-independent environment for reengineering object-oriented systems. In *Proceedings of CoSET '00 (2nd International Symposium on Constructing Software Engineering Tools)*, June 2000.

[Dmi04] Mikhail Dmitriev. Profiling Java applications using code hotswapping and dynamic call graph revelation. In *WOSP '04: Proceedings of the Fourth International Workshop on Software and Performance*, pages 139–150. ACM Press, 2004.

[DRW00] Alastair Dunsmore, Marc Roper, and Murray Wood. Object-oriented inspection in the face of delocalisation. In *Proceedings of ICSE '00 (22nd International Conference on Software Engineering)*, pages 467–476. ACM Press, 2000.

[DS84] L. Peter Deutsch and Allan M. Schiffman. Efficient implementation of the Smalltalk-80 system. In *Proceedings POPL '84*, Salt Lake City, Utah, January 1984.

[FAFH09] Michael Furr, Jong-hoon (David) An, Jeffrey S. Foster, and Michael Hicks. Static type inference for Ruby. In *Proceedings of the 2009 ACM Symposium on Applied Computing*, SAC '09, pages 1859–1866, New York, NY, USA, 2009. ACM.

[Fer95] Mary F. Fernández. Simple and effective link-time optimization of Modula-3 programs. *SIGPLAN Not.*, 30(6):103–115, June 1995.

[FJ89] Brian Foote and Ralph E. Johnson. Reflective facilities in Smalltalk-80. In *Proceedings OOPSLA '89, ACM SIGPLAN Notices*, volume 24, pages 327–336, October 1989.

[Fla06] Cormac Flanagan. Hybrid type checking. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 245–256, New York, NY, USA, 2006. ACM.

[FNT15] Mark Faldborg, Troels Lisberg Nielsen, and Bent Thomsen. Type systems and programmers: A look at optional typing in Dart. Master's thesis, Aalborg University, 2015.

[FS07] Hilaire Fernandes and Serge Stinckwich. Morphic, les interfaces utilisateurs selon Squeak, January 2007.

[Fur09] Michael Furr. *Combining Static and Dynamic Typing in Ruby*. PhD thesis, University of Maryland, 2009.

[GÎ10] Tudor Gîrba. The Moose book, 2010.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley Professional, Reading, Mass., 1995.

[GJ90] Justin O. Graver and Ralph E. Johnson. A type system for Smalltalk. In *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '90, pages 136–150, New York, NY, USA, 1990. ACM.

[GMD+10] Mark Grechanik, Collin McMillan, Luca DeFerrari, Marco Comi, Stefano Crespi, Denys Poshyvanyk, Chen Fu, Qing Xie, and Carlo Ghezzi. An empirical investigation into a large-scale Java open source code repository. In *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '10, pages 11:1–11:10, New York, NY, USA, 2010. ACM.

[GR83] Adele Goldberg and David Robson. *Smalltalk 80: the Language and its Implementation*. Addison Wesley, Reading, Mass., May 1983.

[HBL+14] Emily Hill, David Binkley, Dawn Lawrie, Lori Pollock, and K. Vijay-Shanker. An empirical study of identifier splitting techniques. *Empirical Software Engineering*, 19(6):1754–1780, 2014.

[HCN00] R. Harrison, S. Counsell, and R. Nithi. Experimental assessment of the effect of inheritance on the maintainability of object-oriented systems. *J. Syst. Softw.*, 52(2-3):173–179, June 2000.

[HCU91a] Urs Hölzle, Craig Chambers, and David Ungar. Ecoop'91 european conference on object-oriented programming: Geneva, switzerland, july 15–19, 1991 proceedings. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 21–38, Berlin, Heidelberg, 1991. Springer Berlin Heidelberg.

[HCU91b] Urs Hölzle, Craig Chambers, and David Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In P. America, editor, *Proceedings ECOOP '91*, volume 512 of *LNCS*, pages 21–38, Geneva, Switzerland, July 1991. Springer-Verlag.

[HDN07] Niklaus Haldimann, Marcus Denker, and Oscar Nierstrasz. Practical, pluggable types. In *Proceedings of the 2007 International Conference on Dynamic Languages (ICDL 2007)*, pages 183–204. ACM Digital Library, 2007.

[HG12] Brian Hackett and Shu-yu Guo. Fast and precise hybrid type inference for JavaScript. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 239–250, New York, NY, USA, 2012. ACM.

[HH09] Alex Holkner and James Harland. Evaluating the dynamic behaviour of Python applications. In *Proceedings of the Thirty-Second Australasian Conference on Computer Science-Volume 91*, pages 19–28. Australian Computer Society, Inc., 2009.

[HKR+14] Stefan Hanenberg, Sebastian Kleinschmager, Romain Robbes, Éric Tanter, and Andreas Stefik. An empirical study on the impact of static typing on software maintainability. *Empirical Software Engineering*, 19(5):1335–1382, 2014.

[HLBAL05] A. Hamou-Lhadj, E. Braun, D. Amyot, and T. Lethbridge. Recovering behavioral design models from execution traces. In *Proceedings IEEE European Conference on Software Maintenance and Reengineering (CSMR 2005)*, pages 112–121, Los Alamitos CA, 2005. IEEE Computer Society Press.

[HU94] Urs Hölzle and David Ungar. Optimizing dynamically-dispatched calls with run-time type feedback. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, PLDI '94, pages 326–336, New York, NY, USA, 1994. ACM.

[KBR14] Juraj Kubelka, Alexandre Bergel, and Romain Robbes. Asking and answering questions during a programming change task in the Pharo language. In *Proceedings of the 5th Workshop on Evaluation and Usability of Programming Languages and Tools*, PLATEAU '14, pages 1–11, New York, NY, USA, 2014. ACM.

[KF10] Kenneth Knowles and Cormac Flanagan. Hybrid type checking. *ACM Trans. Program. Lang. Syst.*, 32(2):6:1–6:34, February 2010.

[KHR+12] S. Kleinschmager, S. Hanenberg, R. Robbes, E. Tanter, and A. Stefik. Do static type systems improve the maintainability of software systems? An empirical study. In *2012 IEEE 20th International Conference on Program Comprehension (ICPC)*, pages 153 –162, June 2012.

[Lan92] William Landi. Undecidability of static analysis. *ACM Lett. Program. Lang. Syst.*, 1(4):323–337, December 1992.

[LB94] James R. Larus and Thomas Ball. Rewriting executable files to measure program behavior. *Software Practice & Experience*, 24:197–218, 1994.

[LLS+16] Hui Liu, Qiurong Liu, Cristian-Alexandru Staicu, Michael Pradel, and Yue Luo. Nomen est omen: Exploring and exploiting similarities between argument and parameter names. In *International Conference on Software Engineering (ICSE)*, 2016.

[LR91] William Landi and Barbara G. Ryder. Pointer-induced aliasing: A problem classification. In *Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '91, pages 93–103, New York, NY, USA, 1991. ACM.

[LSS+15] Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J. Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z. Guyer, Uday P. Khedker, Anders Møller, and Dimitrios Vardoulakis. In defense of soundiness: A manifesto. *Commun. ACM*, 58(2):44–46, January 2015.

[MBGN16] Nevena Milojković, Clément Béra, Mohammad Ghafari, and Oscar Nierstrasz. Inferring types by mining class usage frequency from inline caches. In *Proceedings of International Workshop on Smalltalk Technologies (IWST 2016)*, pages 6:1–6:11, 2016.

[MCL+15] Nevena Milojković, Andrea Caracciolo, Mircea Lungu, Oscar Nierstrasz, David Röthlisberger, and Romain Robbes. Polymorphism in the spotlight: Studying its prevalence in Java and Smalltalk. In *Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension*, pages 186–195. IEEE Press, 2015. Published.

[MGN17a] Nevena Milojković, Mohammad Ghafari, and Oscar Nierstrasz. Exploiting type hints in method argument names to improve lightweight type inference. In *25th IEEE International Conference on Program Comprehension*, 2017.

[MGN17b] Nevena Milojković, Mohammad Ghafari, and Oscar Nierstrasz. It's duck (typing) season! In *25th IEEE International Conference on Program Comprehension (ERA Track)*, 2017.

[MH90] D. Mancl and W. Havanas. A study of the impact of C++ on software maintenance. In *Proceedings. Conference on Software Maintenance 1990*, pages 63–69, 1990.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[MMI14] André Murbach Maidl, Fabio Mascarenhas, and Roberto Ierusalimschy. Typed Lua: An optional type system for Lua. In *Proceedings of the Workshop on Dynamic Languages and Applications*, Dyla'14, pages 3:1–3:10, New York, NY, USA, 2014. ACM.

[MN16] Nevena Milojković and Oscar Nierstrasz. Exploring cheap type inference heuristics in dynamically typed languages. In *Proceedings of the 2016 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Onward! 2016*, pages 43–56, New York, NY, USA, 2016. ACM.

[Moo07] Colin Moock. *Essential Actionscript 3.0*. O'Reilly, first edition, 2007.

[MSK07] Martin Madsen, Peter Sørensen, and Kristian Kristensen. Ecstatic–type inference for Ruby using the Cartesian product algorithm. Master's thesis, Aalborg University, 2007.

[NDG05] Oscar Nierstrasz, Stéphane Ducasse, and Tudor Gîrba. The story of Moose: an agile reengineering environment. In *Proceedings of the European Software Engineering Conference (ESEC/FSE'05)*, pages 1–10, New York, NY, USA, September 2005. ACM Press. Invited paper.

[Nys03] Sven-Olof Nyström. A soft-typing system for Erlang. In *Proceedings of the 2003 ACM SIGPLAN Workshop on Erlang*, ERLANG '03, pages 56–71, New York, NY, USA, 2003. ACM.

[Odg14] Morten Passow Odgaard. JavaScript type inference using dynamic analysis. Master's thesis, Aarhus University, 2014.

[Ous98] John K. Ousterhout. Scripting: Higher level programming for the 21st century. *IEEE Computer*, 31(3):23–30, March 1998.

[Pal05] Mike Pall. The LuaJIT Project, 2005. http://luajit.org/.

[PBMH11] Chris Parnin, Christian Bird, and Emerson Murphy-Hill. Java generics adoption: How new features are introduced, championed, or ignored. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, MSR '11, pages 3–12, New York, NY, USA, 2011. ACM.

[PFD11] Daryl Posnett, Vladimir Filkov, and Premkumar Devanbu. Ecological inference in empirical software engineering. In

Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering, ASE '11, pages 362–371, Washington, DC, USA, 2011. IEEE Computer Society.

[Pil04] Mark Pilgrim. *Dive Into Python*. APress, 2004.

[PMW09] Frédéric Pluquet, Antoine Marot, and Roel Wuyts. Fast type reconstruction for dynamically typed programming languages. In *DLS '09: Proceedings of the 5th Symposium on Dynamic languages*, pages 69–78, New York, NY, USA, 2009. ACM.

[PS91] Jens Palsberg and Michael I. Schwartzbach. Object-oriented type inference. In *Proceedings OOPSLA '91, ACM SIGPLAN Notices*, volume 26, pages 146–161, November 1991.

[PSS15] M. Pradel, P. Schuh, and K. Sen. TypeDevil: Dynamic type inconsistency analysis for JavaScript. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 314–324, 2015.

[PTP07] Guillaume Pothier, Éric Tanter, and José Piquer. Scalable omniscient debugging. *Proceedings of the 22nd Annual SCM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'07)*, 42(10):535–552, 2007.

[PVC01] Michael Paleczny, Christopher Vick, and Cliff Click. The Java hotspot^{TM} server compiler. In *Proceedings of the 2001 Symposium on Java^{TM} Virtual Machine Research and Technology Symposium - Volume 1*, JVM'01, pages 1–1, Berkeley, CA, USA, 2001. USENIX Association.

[PW88] Lewis J. Pinson and Richard S. Wiener. *Objective-C*. Addison Wesley, 1988.

[Ram94] G. Ramalingam. The undecidability of aliasing. *ACM Trans. Program. Lang. Syst.*, 16(5):1467–1471, September 1994.

[RBFDD98] Pascal Rapicault, Mireille Blay-Fornarino, Stéphane Ducasse, and Anne-Marie Dery. Dynamic type inference

to support object-oriented reengineering in Smalltalk. In *Proceedings of the ECOOP '98 International Workshop Experiences in Object-Oriented Reengineering, abstract in Object-Oriented Technology (ECOOP '98 Workshop Reader forthcoming LNCS)*, pages 76–77, 1998.

[RCH12] Aseem Rastogi, Avik Chaudhuri, and Basil Hosmer. The ins and outs of gradual type inference. *SIGPLAN Not.*, 47(1):481–494, January 2012.

[RHBV11] Gregor Richards, Christian Hammer, Brian Burg, and Jan Vitek. The eval that men do: A large-scale study of the use of eval in JavaScript applications. In *Proceedings of the 25th European Conference on Object-oriented Programming*, ECOOP'11, pages 52–78, Berlin, Heidelberg, 2011. Springer-Verlag.

[RHV+09] David Röthlisberger, Marcel Härry, Alex Villazón, Danilo Ansaloni, Walter Binder, Oscar Nierstrasz, and Philippe Moret. Augmenting static source views in IDEs with dynamic metrics. In *Proceedings of the 25th International Conference on Software Maintenance (ICSM 2009)*, pages 253–262, Los Alamitos, CA, USA, 2009. IEEE Computer Society.

[RHV+11] David Röthlisberger, Marcel Härry, Alex Villazón, Danilo Ansaloni, Walter Binder, Oscar Nierstrasz, and Philippe Moret. Exploiting dynamic information in IDEs improves speed and correctness of software maintenance tasks. *Transactions on Software Engineering*, 2011.

[RKG04] Atanas Rountev, Scott Kagan, and Michael Gibas. Evaluating the imprecision of static analysis. In *PASTE '04: Proceedings of the 5th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 14–16, New York, NY, USA, 2004. ACM.

[RL08] R. Robbes and M. Lanza. How program history can improve code completion. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ASE '08, pages 317–326, Washington, DC, USA, 2008. IEEE Computer Society.

[RLBV10] Gregor Richards, Sylvain Lebresne, Brian Burg, and Jan Vitek. An analysis of the dynamic behavior of JavaScript programs. *SIGPLAN Not.*, 45(6):1–12, June 2010.

[RLR12] Romain Robbes, Mircea Lungu, and David Roethlisberger. How do developers react to API deprecation? The case of a Smalltalk ecosystem. In *Proceedings of the 20th International Symposium on the Foundations of Software Engineering (FSE'12)*, pages 56:1 – 56:11, 2012.

[RMR03] Atanas Rountev, Ana Milanova, and Barbara G. Ryder. Fragment class analysis for testing of polymorphism in Java software. In *ICSE '03: Proceedings of the 25th IEEE International Conference on Software Engineering*, pages 210–220, Los Alamitos, CA, USA, 2003. IEEE Computer Society Press.

[RND09] David Röthlisberger, Oscar Nierstrasz, and Stéphane Ducasse. Autumn leaves: Curing the window plague in IDEs. In *Proceedings of the 16th Working Conference on Reverse Engineering (WCRE 2009)*, pages 237–246, Los Alamitos, CA, USA, 2009. IEEE Computer Society.

[RNDB09] David Röthlisberger, Oscar Nierstrasz, Stéphane Ducasse, and Alexandre Bergel. Tackling software navigation issues of the Smalltalk IDE. In *Proceedings of International Workshop on Smalltalk Technologies (IWST 2009)*, pages 58–67, New York, NY, USA, 2009. ACM.

[Sal04] Michael Salib. Faster than C: Static type inference with Starkiller. In *in PyCon Proceedings, Washington DC*, pages 2–26. SpringerVerlag, 2004.

[SES05] Janice Singer, Robert Elves, and Margaret-Anne Storey. NavTracks: Supporting navigation in software maintenance. In *International Conference on Software Maintenance (ICSM'05)*, pages 325–335, Washington, DC, USA, sep 2005. IEEE Computer Society.

[SF14] Carlos Souza and Eduardo Figueiredo. How do programmers use optional typing? An empirical study. In *Proceedings of*

the 13th International Conference on Modularity, MODULARITY '14, pages 109–120, New York, NY, USA, 2014. ACM.

[SH14] Samuel Spiza and Stefan Hanenberg. Type names without static type checking already improve the usability of APIs (as long as the type names are correct): An empirical study. In *Proceedings of the 13th International Conference on Modularity*, MODULARITY '14, pages 99–108, New York, NY, USA, 2014. ACM.

[SLN14] Boris Spasojević, Mircea Lungu, and Oscar Nierstrasz. Mining the ecosystem to improve type inference for dynamically typed languages. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! '14, pages 133–142, New York, NY, USA, 2014. ACM.

[SLN16] Boris Spasojević, Mircea Lungu, and Oscar Nierstrasz. A case study on type hints in method argument names in Pharo Smalltalk projects. In *Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 283–292, March 2016.

[SMDV08] Jonathan Sillito, Gail C. Murphy, and Kris De Volder. Asking and answering questions during a programming change task. *IEEE Trans. Softw. Eng.*, 34:434–451, July 2008.

[Smi] Rob Smit. Pegon. https://sourceforge.net/projects/pegon/.

[SS04] S. Alexander Spoon and Olin Shivers. Demand-driven type inference with subgoal pruning: Trading precision for scalability. In *Proceedings of ECOOP'04*, pages 51–74, 2004.

[SS05] S. Alexander Spoon and Olin Shivers. Dynamic data polyvariance using source-tagged classes. In Roel Wuyts, editor, *Proceedings of the Dynamic Languages Symposium'05*, pages 35–48. ACM Digital Library, 2005.

[ST06] Jeremy G. Siek and Walid Taha. Gradual typing for functional languages. In *Proceedings, Scheme and Functional*

Programming Workshop 2006, pages 81–92. University of Chicago TR-2006-06, 2006.

[ST07] Jeremy Siek and Walid Taha. Gradual typing for objects. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP'07)*, volume 4609 of *LNCS*, pages 151–175. Springer Verlag, 2007.

[Str] The Strongtalk type system for Smalltalk. http://bracha.org/nwst.html.

[Sub13] V. Subramaniam. *Programming Groovy 2: Dynamic Productivity for the Java Developer*. Number v. 2 in Pragmatic Bookshelf. Pragmatic Bookshelf, 2013.

[Sus97] Horwitz Susan. Precise flow-insensitive may-alias analysis is NP-hard. *ACM Trans. Program. Lang. Syst.*, 19(1):1–6, January 1997.

[TAD+10] E. Tempero, C. Anslow, J. Dietrich, T. Han, Jing Li, M. Lumpe, H. Melton, and J. Noble. The Qualitas Corpus: A curated collection of Java code for empirical studies. In *Software Engineering Conference (APSEC), 2010 17th Asia Pacific*, pages 336 –345, December 2010.

[TF11] Sam Tobin-Hochstadt and Matthias Felleisen. The design and implementation of Typed Scheme: From scripts to programs. *CoRR*, abs/1106.2575, 2011.

[TFH09] Dave Thomas, Chad Fowler, and Andy Hunt. *Programming Ruby 1.9: The Pragmatic Programmers' Guide*. Pragmatic Bookshelf, 3rd edition, 2009.

[THSA] S. Tobin-Hochstadt and V. St-Amour. The typed Racket guide. http://docs.racket-lang.org/ts-guide/.

[TNM08] Ewan Tempero, James Noble, and Hayden Melton. How do Java programs use inheritance? An empirical study of inheritance in Java software. In *Proceedings of the 22Nd European Conference on Object-Oriented Programming*, ECOOP '08, pages 667–691, Berlin, Heidelberg, 2008. Springer-Verlag.

[TP00] Frank Tip and Jens Palsberg. Scalable propagation-based call graph construction algorithms. In *Proceedings of the 15th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '00, pages 281–293, New York, NY, USA, 2000. ACM.

[US87] David Ungar and Randall B. Smith. Self: The power of simplicity. In *Proceedings OOPSLA '87, ACM SIGPLAN Notices*, volume 22, pages 227–242, December 1987.

[VKSB14] Michael M. Vitousek, Andrew M. Kent, Jeremy G. Siek, and Jim Baker. Design and evaluation of gradual typing for Python. *SIGPLAN Not.*, 50(2):45–56, October 2014.

[VS16] Michael M Vitousek and Jeremy G Siek. From optional to gradual typing via transient checks. In *5th Script To Program Evolution Workshop*, 2016.

[WF07] Philip Wadler and Robert Bruce Findler. Well-typed programs can't be blamed. In *Proceedings of the Workshop on Scheme and Functional Programming*, pages 15–26, 2007.

[WH92] Norman Wilde and Ross Huitt. Maintenance support for object-oriented programs. *IEEE Transactions on Software Engineering*, SE-18(12):1038–1044, December 1992.

[WHH11] A. Wiese, V. Ho, and E. Hill. A comparison of stemmers on source code identifiers for software search. In *2011 27th IEEE International Conference on Software Maintenance (ICSM)*, pages 496–499, September 2011.

[WS01] Tiejun Wang and Scott F. Smith. Precise constraint-based type inference for Java. In G. Goos, J. Hartmanis, and J. van Leeuwen, editors, *Proceedings ECOOP '01*, volume 2072 of *LNCS*, pages 99–118, Budapest, Hungary, June 2001. Springer-Verlag.

[XZC+16] Zhaogui Xu, Xiangyu Zhang, Lin Chen, Kexin Pei, and Baowen Xu. Python probabilistic type inference with natural language support. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 607–618, New York, NY, USA, 2016. ACM.

[Zan13] Matt Zandstra. *PHP Objects, Patterns, and Practice*. Apress, Berkely, CA, USA, 4th edition, 2013.